

NO-A184 873

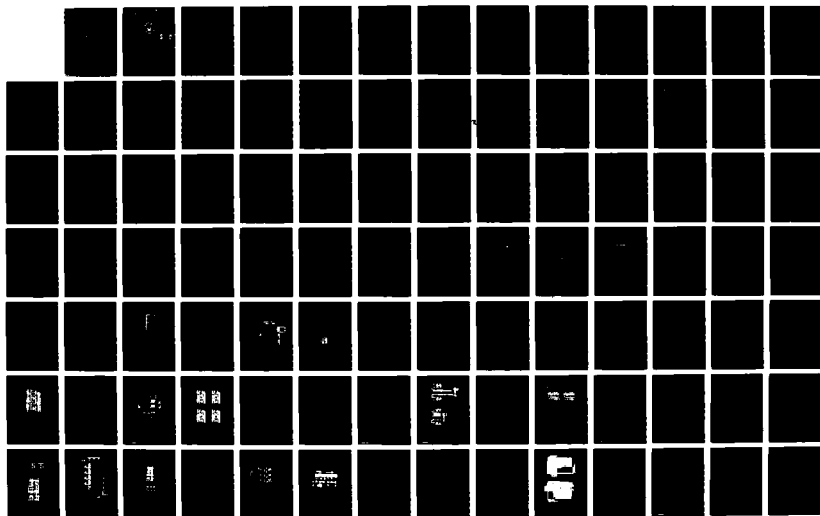
TECHNOLOGY UPGRADE OF A SILICON COMPILER(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA E G MALAGON JUN 87

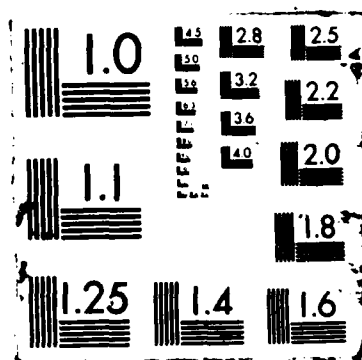
1/2

UNCLASSIFIED

F/G 12/5

NL





AD-A184 873

DTIC FILE COPY

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
SEP 29 1987
S D

THESIS

TECHNOLOGY UPGRADE OF A SILICON COMPILER

by

Eva G. Malagon

June 1987

Thesis Advisor

D. Kirk

Approved for public release; distribution is unlimited.

015

ADA184873

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (if applicable) 62		7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000			
8a NAME OF FUNDING/SPONSORING ORGANIZATION		8b OFFICE SYMBOL (if applicable)		9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO		PROJECT NO	TASK NO
					WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification) TECHNOLOGY UPGRADE OF A SILICON COMPILER					
12 PERSONAL AUTHOR(S) Malagon, Eva G.					
13a TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM TO		14 DATE OF REPORT (Year Month Day) 1987 June	
				15 PAGE COUNT 119	
16 SUPPLEMENTARY NOTATION					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
			silicon compilation, vlsi design, integrated circuit design, NMOS, SCMOS		
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>A 1.5 micron dual layer metal scaleable CMOS standard cell library is inserted into the NMOS based silicon compiler, MacPitts. The MacPitts data-path consisting of a collection of registers and arithmetic/logic units (organelles) and sequenced by a three phase clock was modified to accept two phase clocking and SCMOS organelles.</p>					
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION unclassified		
22a NAME OF RESPONSIBLE INDIVIDUAL Kirk, D.			22b TELEPHONE (Include Area Code) 408-646-3905		22c OFFICE SYMBOL 62KI

Approved for public release; distribution is unlimited.

Technology Upgrade of a Silicon Compiler

by

Eva G. Malagon
Captain, United States Marine Corps
B.S., Illinois Institute of Technology, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

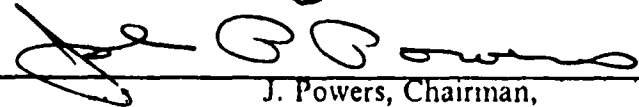
Author:

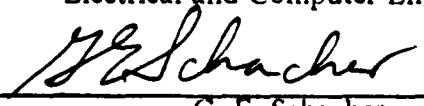

Eva G. Malagon

Approved by:


D. Kirk, Thesis Advisor


M. Zyda, Second Reader


J. Powers, Chairman,
Electrical and Computer Engineering


G. E. Schacher,
Dean of Science and Engineering

ABSTRACT

A 1.5 micron dual layer metal scaleable CMOS standard cell library is inserted into the NMOS based silicon compiler, MacPitts. The MacPitts data-path consisting of a collection of registers and arithmetic/logic units (organelles) and sequenced by a three phase clock was modified to accept two phase clocking and SCMOS organelles.



Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	11
A.	BACKGROUND	11
B.	CODING	12
C.	SCOPE	13
D.	NOTATION	13
II.	MACPITTS SYSTEM OVERVIEW	15
A.	INTRODUCTION	15
B.	HIGH LEVEL DESCRIPTION	15
C.	INTERMEDIATE LEVEL DESCRIPTION	15
D.	LOWER LEVEL DESCRIPTION	16
E.	COORDINATING MACPITTS PROGRAMS THROUGH THE MAKEFILE	16
F.	MACPITTS DESIGN ORGANIZATION	18
	1. Top-Level Design	18
	2. Second Level Design	18
	3. Internal Level of Design	21
	4. Conversion Procedure	25
	5. Key Constructs	29
III.	DATA-PATH LAYOUT	32
A.	METHODOLOGY	32
	1. Defstructs	33
B.	ARCHITECTURAL IMPLEMENTATION OF A UNIT	36
	1. Database	36
C.	ORGANELLE-LIST	37
	1. Data-Path Placement Algorithm	37
	2. Layout Organelle	38
	3. Tail	40

	4. Drive	43
D.	ORGANELLE LAYOUT	44
	1. Layout Geometry of an Organelle	45
	2. Organelle Extensions	47
	3. Layout Gen-Form	51
E.	PORTS	51
	1. Port-Input	51
	2. Port Output	54
	3. Port-Internal	54
F.	DATA-PATH LAYOUT	55
G.	SUMMARY	56
IV.	SCMOS DATA-PATH	60
A.	SOURCE PROGRAMS CONTAINING ORGANELLES	60
B.	MODIFICATION OF AN ORGANELLE	62
	1. Insertion of a SCMOS Adder Organelle	64
	2. Organelle Library Changes	65
C.	SCMOS CELL LIBRARY ATTRIBUTES	72
D.	ORGANELLES.L	72
	1. Two Input Ripple Adder with Carry	72
	2. Two input Equality (EQU and =) Cells	73
E.	DATA-PATH.L	76
	1. Static D-Flip-Flop Memory Element (Register)	76
	2. Multiplexer	86
F.	FRAME.L	88
	1. Two-Phase Clocking	88
G.	GENERAL.L	88
	1. SuperBuffers	88
	2. River Router	89
H.	LAYER CONVERSION	89
I.	SUMMARY	89
V.	AREAS FOR FURTHER DEVELOPMENT	96
A.	SUGGESTED CHANGES TO SOURCE PROGRAMS	96
	1. Lincoln.l	96

2.	L5.l	96
3.	Defstructs.l	97
4.	Library	98
5.	Organelles.l	99
6.	Data-Path.l	99
7.	Frame.l	100
8.	General.l	102
9.	Extract.l	102
10.	Prepass.l	102
11.	Control and Order.l	103
12.	Pads.l	103
13.	Flags.l	106
B.	TECHNOLOGY INDEPENDENCE	106
C.	ROUTING AND PLACEMENT	106
VI.	CONCLUSIONS	108
APPENDIX:	CIFDEF.L	110
LIST OF REFERENCES	114
BIBLIOGRAPHY	115
INITIAL DISTRIBUTION LIST	117

LIST OF TABLES

1. GEN-FORM	25
2. MACPITTS INPUT PROGRAM	27
3. OBJECT FILE	28
4. INPUT PROGRAM FOR A 1-BIT NAND CIRCUIT	33
5. OBJECT FILE FOR A NAND CIRCUIT	34
6. SCMOS NAND ORGANELLE GEN-FORM	41
7. ARGUMENTS TO THE GEN-FORM	43
8. CODING OF SCMOS NAND ORGANELLE	46
9. CODING FOR THE RIVER ROUTER	53
10. ORGANELLE I/O REQUIREMENTS	60
11. CIF TO L5 CODE	63
12. ADDER LIBRARY FUNCTION	67
13. MACPITTS INPUT PROGRAM FOR AN ADDER CHIP	69
14. MACPITTS INPUT PROGRAM FOR A REGISTER	85
15. TAXI METER INPUT PROGRAM	90
16. TAXI CHIP DATAPATH SPECIFICATION	92
17. SCMOS CELLS AND MACPITTS REQUIREMENTS	95

LIST OF FIGURES

2.1	Top-Level Layout Constraint	19
2.2	Ring-Layout Constraint	20
2.3	Skeleton Power/Ground/Clock Distribution Bus	21
2.4	Internal Layout	22
2.5	Unit Layout of a Datapath	23
2.6	Organelle Functional Block	26
2.7	Item Topology	30
2.8	Hierarchical Construction of a MacPitts Circuit	31
3.1	Flowchart of Functions in Datapath	37
3.2	Functional Layout of the Datapath	38
3.3	Unit Placement	39
3.4	Bounding Box for a Nand Organelle	42
3.5	Bit-slice Unit Topology	45
3.6	Output-Extension	48
3.7	Lower Extension	49
3.8	Upper Extension	50
3.9	Port Input Topological Layout	52
3.10	Stipple Plot of a River Connection	53
3.11	Port-Output Unit	55
3.12	Topological Layout of an NMOS 2:1 Multiplexer	56
3.13	MacPitts Generated Hybrid Nand Circuit	58
3.14	Datapath containing SCMOS Organelle	59
4.1	SCMOS Adderbit1 Terminal Locations	68
4.2	MacPitts generated SCMOS Adder Chip	70
4.3	Windowed Plot of 2-bit wide SCMOS Adder Datapath	71
4.4	Topology of Adder Chip	74
4.5	Comparison of NMOS and HYBRID Adder Circuits	75
4.6	Mask Layout of a SCMOS Equality (EQU) Gate	77

4.7	Sequencer Topology	78
4.8	Sequencer with Counter and Stack	79
4.9	MacPitts Hardware Implementing a Finite State Machine	80
4.10	Bit-Unit Topology for a Bit-List of (1 0)	81
4.11	Comparison of an NMOS and SCMOS MacPitts Register	82
4.12	Hybrid Circuit with SCMOS Register	83
4.13	Windowed SCMOS Register, NMOS Clock Driver and Mux	84
4.14	Topology of an Equality Circuit with State Register	85
4.15	Mask Layout of NMOS Clock Driver	86
4.16	SCMOS 2:1 Multiplexer	87
4.17	Comparison of an NMOS and SCMOS Taxi Meter Chip	91
4.18	Topological Layout of the Taxi Chip Data-Path	93

ACKNOWLEDGEMENTS

Appreciation is expressed to Professor D. Kirk for the time spent in guidance and encouragement in the pursuit of this thesis. Of great assistance where R. Limes and A. Wong in working out systems level problems on the VAX 11/780 system. My thanks to LtCdr. M. A. Malagon-Fajar for providing the motivation and encouragement to work in the area of VLSI design. My deepest thanks to my father, Mr. H. Spelter IV, who provided years of nurturing and support and an intellectual environment that formed the basis for academic achievement and growth. My thanks to my grandmother, Mrs. Josephine Szekrenyi for her assistance with my daughter Carolina Malagon during the final months of this thesis without whom the completion of this thesis would have been much hindered. My thanks, also, to the Kupeli family who provided tremendous assistance to me, soon after the birth of my daughter thus allowing me to continue my education uninterrupted. My thanks to the United States Marine Corps for the opportunity to pursue a higher education at the Naval Postgraduate School.

This thesis is dedicated to the American People in the pursuit and maintenance of FREEDOM.

I. INTRODUCTION

The *MacPitts silicon compiler* is a set of VLSI CAD software programs whose function is to automatically generate a circuit layout from a user-specified behavioral description of the circuit in the form of an input program. MacPitts is N-channel (NMOS) technology based. With alternate technologies and design rules available, e.g., scalable Complementary Metal Oxide (SCMOS), CMOS-PW, and Gallium arsenide (GaAs), a method of compiling a design in those technologies is desirable to take advantage of high speed, low power, reduced chip area and higher reliability.

A. BACKGROUND

MacPitts was developed by researchers at Massachusetts Institute of Technology Lincoln Laboratories in the early 1980's under a contract to DARPA. At the time of its implementation, 4 micron NMOS processes were the state of the art technology. The advanced graphics layout editor of that time was CAESAR. Technology has radically changed since then, and SCMOS technology with its low power consumption and smaller feature size of 3 microns and below is a preferred technology today, and CAESAR has evolved into Magic, a more advanced graphics layout editor that supports SCMOS designs. The problem of making a silicon compiler dynamic enough to support rapid technology changes is the motivation behind the research on MacPitts.

MacPitts, as a VLSI research tool, was initially installed on the Naval Postgraduate School, Monterey, California, VAX-11/780 computer facility by D. Carlson, 1984. Carlson's research recommended structural modifications to the MacPitts system: CMOS layouts which would involve writing a new cell library and modifying the control unit architecture. A. Froede, 1985, outlined the basic building blocks that the MacPitts compiler used in building its circuits: data-path, flags, sequencer, and the Weinberger array control unit. Froede's recommendations included the addition of NMOS superbuffers to all input and output lines from the data-path, sequencer and flags, and to all clock lines driving registers and flags; redesign of the frame to allow pads on four sides of the chip rather than the current three sides; implementation of a two phase clocking scheme instead of the MacPitts three phase; a redesign of the data-path to allow data to enter or leave from either the left or right

side to reduce length of wire runs from the pads; and an alternate placement routine whereby the flags module is not appended to the end of the data-path module thus extending the length of the chip excessively. Further study of the interrelationship between algorithmic syntax used to write the input program, identified by a .mac extension, and the resulting circuit layout was done by R. Larrabee, 1985.

To employ the recommended changes, an understanding of the MacPitts source code is necessary. A study of MacPitts compiler organization and layout language by M. A. Malagon-Fajar, 1986, set the basis for an understanding of the lists and structures used by the MacPitts compiler to design a circuit. [Ref. 1] An NMOS equality (=) test cell designed by A. Mullarky, was inserted into the NMOS cell library of the MacPitts compiler replacing the original hierarchically designed NMOS equality (=) cell. The resultant MacPitts-generated NMOS circuit was considerable smaller in size.

A standard set of CMOS arithmetic/logic and memory cells was also designed by A. Mullarky. [Ref. 2] The clocked SCMOS memory cells were designed for two phase clocking. The work reported in this thesis inserted the SCMOS cells into the NMOS technology dependent layout generation routines of MacPitts and altered the MacPitts code to generate a two phase clock. A change of technology from NMOS to SCMOS introduced a significant change in the controller. The Weinberger array control unit is optimum for NMOS layout strategy but not for SCMOS. Therefore, a microprogrammed controller was designed by J. Harmon, 1987 for insertion into MacPitts. [Ref. 3]

B. CODING

MacPitts is a complex software system containing 1.5 megabytes of binary executable code. The source code of MacPitts is written in Franz Lisp Opus 38.9. The compiler can be separated into a set of programs that produce a technology independent structural specification of the circuit called an object file and a set of technology dependent routines that produce the actual layout of the circuit and output a description of the circuit in the low level language CalTech Intermediate Form (CIF). The former is the front-end of the compiler, the latter is the back-end. A silicon compiler is similar to a language compiler, except that the object file does not produce machine code but rather an intermediate description of the circuit.

Lisp is a symbolic computation language suitable for representing complex systems in terms of symbols and structures. The process is likened to a VLSI design

engineer who uses symbols to represent standard cells or macros placed in the floorplan of a circuit. Just as a designer can move the symbol for a cell on the floorplan, the compiler can move a symbol from one place to another. The design of a VLSI circuit is a list-building process in which the lowest level symbols are the layout geometries of arithmetic/logic/memory units. Lists and structures are used to combine symbols into more complex entities evolving into a chip. Lisp is a block structured language, meaning functions are nested inside other functions. This allows ease of tracing code. The Franz Lisp stepper package and its debugging routines facilitate easier writing and reading of code.

C. SCOPE

A partial CMOS cell library was inserted into the data-path, and a controller was designed for the SC MOS version of MacPitts. This thesis explains the pad placement algorithm and changes the algorithm to allow pads on all sides of the chip. To complete the placement of pads on four sides the algorithm generating the wiring from the pads to the controller and data-path must also be changed. The algorithm for floorplan placement is exposed allowing for future work on a new placement algorithm for MacPitts.

Alteration of the MacPitts compiler code required thorough familiarity with the syntax and contents of the source code. Chapter II discusses the general methodology for the technology modification of the MacPitts data-path. In Chapter III, the intermediate technology independent code describing a simple nand chip is committed to a layout and a bit-slice of a data-path is laid out from the specification in the object file. Chapter IV describes the individual SC MOS cells inserted into MacPitts. Chapter V describes areas for further development. Many tasks, ranging from the insertion of the remainder of the SC MOS cells into MacPitts to a smarter routing and placement algorithm that is compatible with SC MOS requirements and capabilities are discussed.

D. NOTATION

The notation used throughout this thesis is based on the Backus Naur Form (BNF) syntactic notation. Special symbols used in BNF are

::- means "is defined as"

means "or"

***** means "sequence of zero or more"

The Lisp environment is designated by the LISP prompt sign ->. The parenthesis surrounding a LISP form indicates the start and stop of LISP's interpretation of an instruction (..). The UNIX ¹ environment is indicated by a % prompt sign.

¹UNIX is a registered trademark of Bell Laboratories.

II. MACPITTS SYSTEM OVERVIEW

A. INTRODUCTION

There are three levels of description in the MacPitts system: the high level description given by the user in the .mac file (a list of the required elements and processes to be performed by the circuit), which is transformed by the high-level processor or compiler into an intermediate-level description. The compiler then generates the layout of the circuit by generating a description list that is converted into a CIF file for graphical layout and for fabrication using the MOSIS process.

B. HIGH LEVEL DESCRIPTION

The .mac file contains a functional description of the circuit to be designed. The program is a straightforward list of type and number of pads, registers, signals, ports, constants and the processes to be performed. The format for writing the input algorithm is defined by the Backus Naur Form (BNF) grammar in the BNF file. The BNF grammar for cell definition and operation is known by the compiler.

C. INTERMEDIATE LEVEL DESCRIPTION

The intermediate form is a list of sources, destinations, circuit size in number of bits, clock, power, ground, port, signals and their pin numbers. This list of components and the boolean logic equations defining the controller's logic are described as follows:

```
object ::= ((definitions) ::= sources, destinations, ports, signals, registers,  
                                     power, ground, clock, constants  
      (flags)  
      (datapath) ::= units  
      (control) ::= wires and operands  
      (pins) ::= input, output, clock, power, ground, tri-state, I/O.
```

This file is output by the compiler as a .obj file. It is a technology independent description of the circuit. The compiler is located in the higher level routine known as prepass.l and coordinates the seventeen programs that make up the MacPitts silicon compiler. The higher level routines coordinated by the compiler are prepass.l, and extract.l. The compiler then transfers control to the top level layout routine in frame.l to layout the circuit. The routine returns a list containing the chip's geometric layout.

The module generators coordinated by the routine `frame.l` include `control.l`, `data-path.l`, `organelles.l`, `flags.l`, and `pads.l`, all of which are highly technology dependent. In the unmodified compiler, all layouts are described in N-channel (NMOS) technology. A limitation of this silicon compiler is the necessity of redesigning a large portion of code when the technology changes.

The compiler evaluates the global electrical characteristics of the circuit and prints to the screen *statistics* relevant to the circuit. If the *herald* option is selected in the command line, the compiler also prints to the screen messages to inform the designer of each step in the compilation process.

The above routines are highly dependent on the *defstruct* construct defined in the file `lincoln.l` and enumerated in `defstructs.l`. This construct forms the basis for the creation, selection, alteration (mutation) and query (interrogation) of the data required to process the input program, the `.mac` file, and produce a layout of the circuit, an `(.obj)` and `(.cif)` file.

D. LOWER LEVEL DESCRIPTION

A realization of the datapath and control unit of a microprocessor-like circuit is accomplished through the `frame.l` function (layout object). The program `L5.l` is the basis for the layout of the cell in terms of providing functions that allow manipulation of the cell and position of the circuit. This program produces the `cif` description of the cell and also has the capability of reading CAESAR files and converting them to `L5` formatted code. The revision 7 version of `L5` utilized in the Lincoln Boolean Synthesizer (LBS) programs contains an interface to the CAESAR layout graphics editor. The program interfaces cells in CAESAR format (rectangular geometry) to `L5` format (lists of rectangular geometry). As part of this thesis project an interface program was written for the more universal CIF (box geometry) input and is listed in the Appendix.

E. COORDINATING MACPITTS PROGRAMS THROUGH THE MAKEFILE

Each separate program in MacPitts is converted to machine code via the lisp compiler *liszr*. The basic programs initially compiled are `c-routines.c`, `lincoln.l`, `L5.l` and `defstruct.l`. `Lincoln.l` is fundamental to the remainder of the MacPitts system. It contains basic Franz Lisp functions and other functions written to expand the MacPitts system capability to manipulate lists. `Lincoln.l` contains the important *defstructs* package which forms the basis for making items, objects and layouts, for

selecting their parameters, for changing those parameters and for querying the items. The machine coded programs, `lincoln.o`, `defstructs.o` and `L5.o` are loaded into the lisp environment under the file `front-page.l`. This file is included in each subsequent program. These subsequent programs are dependent upon the ability to correctly load or `fasl` the above three programs before they can be machine encoded into a `.o` file. The remainder of the MacPitts source code is compiled and loaded into an executable module called **macpitts**. During the *make* compilation, the files, `rinout` and `pad20b` have their CIF code converted into L5 code and loaded into the `pads.l` program. These two files contain the NMOS 4 and 5 micron pads in their CIF form.

To create an executable files, the `.o` programs are loaded or 'fasled' into the lisp environment along with global variables setting the `macpitts-directory` path name, the user-top-level, the interrupt handler and the default minimum-feature-size. The lisp function (*dumplisp macpitts*) dumps the MacPitts binary code into a lisp environment and saves the environment called `macpitts`. A lisp environment can be invoked by typing *macpitts*. The user will have access not only to Franz Lisp primitives but all the functions defined by the MacPitts source programs. The executable file is created using the UNIX command *make*. Make executes commands in the Makefile to update one or more programs that have been modified since the last creation of the file. The command **make macpitts** begins the timely process of compiling each program and loading those machine encoded programs into the executable file `macpitts`. Since the cell libraries (`organelles.l` and `library`) are loaded into Macpitts only at run time, changes can be made to both files without remaking the executable file. Since changes were being made to built-in organelles as well as the organelle library, the programs `frame.l` and `data-path.l` were removed from the executable `macpitts` file, and also loaded at run time.

To load programs at run time, the library file contains the command

```
(eval compile (boot (concat macpitts-directory '/organelles)))
```

The function *boot* is defined in `lincoln.l` and allows loading of programs at run time. A simpler way to test changes to source code is simply to enter the `macpitts` environment and load the altered functions. Lisp processes the most recent functions. This allows thorough testing of software changes before they are incorporated into the MacPitts baseline program.

F. MACPITTS DESIGN ORGANIZATION

1. Top-Level Design

MacPitts-produced circuit designs are fixed in topology with a standard control and datapath paradigm. At the top-level, the floorplan consists of three symbols, the pins-layout, internal-layout and ring-layout. The pads and their associated ground and power distribution bus comprise the pins-layout. The internal-layout is an invisible box containing the controller and datapath. The rectangular channel formed between the pins and the internal-layout is the wiring channel that connects the pads to the internal layout.

Constraints are readily apparent at this level of the floorplan. Pads are placed on three sides: the top, right and bottom edges of the circuit. The left edge of the circuit is the side without pads. The top of the circuit is identified by the ground pad and the bottom of the circuit is identified by the power pad. The connections to the internal layout are only on the left edge of the internal circuitry. This necessitates long wiring runs around the channel. A less obvious constraint in the floorplan is the weight that the dimensions of the internal-layout and pads have on the wiring channel. If the pad lengths on either top, right or bottom side of the circuit is shorter than the length or width of the internal-layout, the wiring channel or ring will form around the internal-layout. If the internal-layout is smaller in length or width than the length of the pads on a side of the circuit, then the pads length on that side will determine the ring size. Figure 2.1 illustrates the first case and Figure 2.2 illustrates the second case. In the first case, a very long datapath and short controller will result in empty silicon space inside the internal-layout since the largest element in the internal-layout determines the size of the invisible box surrounding it. If the internal-layout is smaller than the length of the pads, the pads determine the size of the invisible box around which the ring is formed. This topology will have significant impact as technology processes shrink and pads in those processes remain relatively large.

2. Second Level Design

The second level or internal-layout is composed of a controller, a datapath and flags block and a routing channel between the two. The controller is located at the bottom left edge of the invisible box surrounding the internal-layout. It is termed the bottom-part of the layout. The top-part of the layout is composed of the datapath followed by the flags block. A power/ground/clock distribution bus called the skeleton embeds the entire internal-layout. Figure 2.3 shows a stipple plot of the skeleton

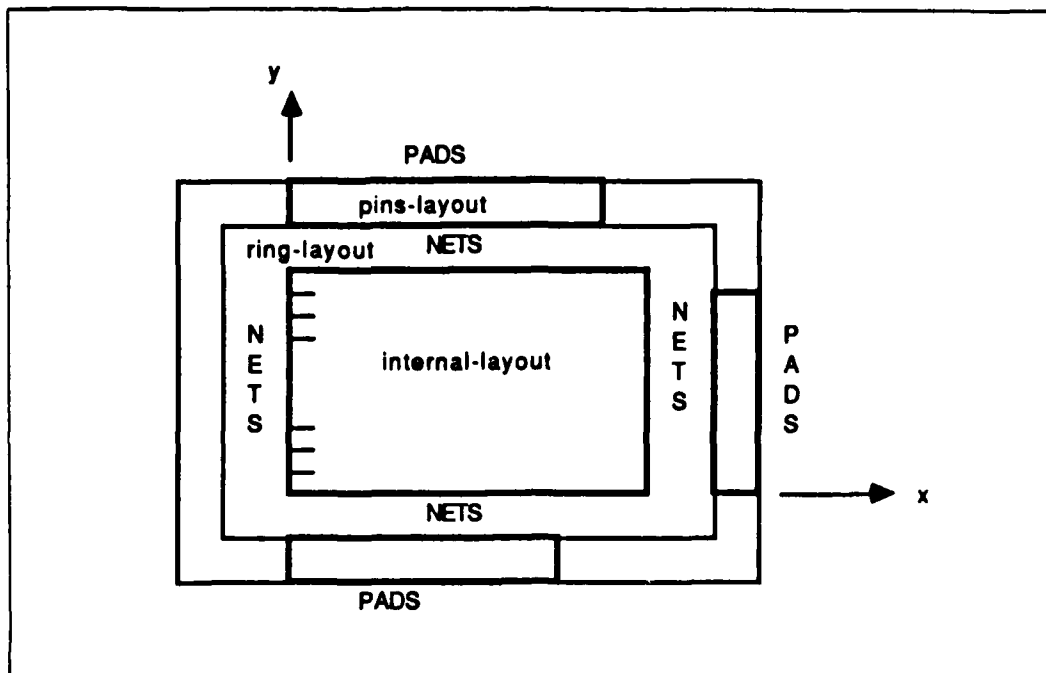


Figure 2.1 Top-Level Layout Constraint.

layout. The clock wires run horizontally through the length of the skeleton. The skeleton receives its power from the power pad on the bottom of the circuit and ground from the ground pad on the top of the circuit. Both pads generate single metal wires that run vertically to butt up to the skeleton. If the skeleton is not long enough to reach the power or ground pad, the power connection is never made. Between the control and datapath/flags block is a horizontal wiring channel. Input/output wiring to the controller is called the wing layout. Internal wiring between the controller and datapath/flags block is automatically accomplished by the river router facility. Between the wiring channel and the datapath, clock wires and drivers are placed. The three-phase clocking scheme generates three clock wires and three-phase clock drivers. A two-phase clocking scheme involved the removal of a clock wire and redesign of the clock driver to a two-phase driver to provide clock signals to the registers. This datapath layout is illustrated in Figure 2.4.

The pads are placed on the frame relative to the internal-layout. The lower left corner of the internal-layout provides the origin of the chip layout from which all pads and wires are positioned.. The width of the internal layout is called the intended-top

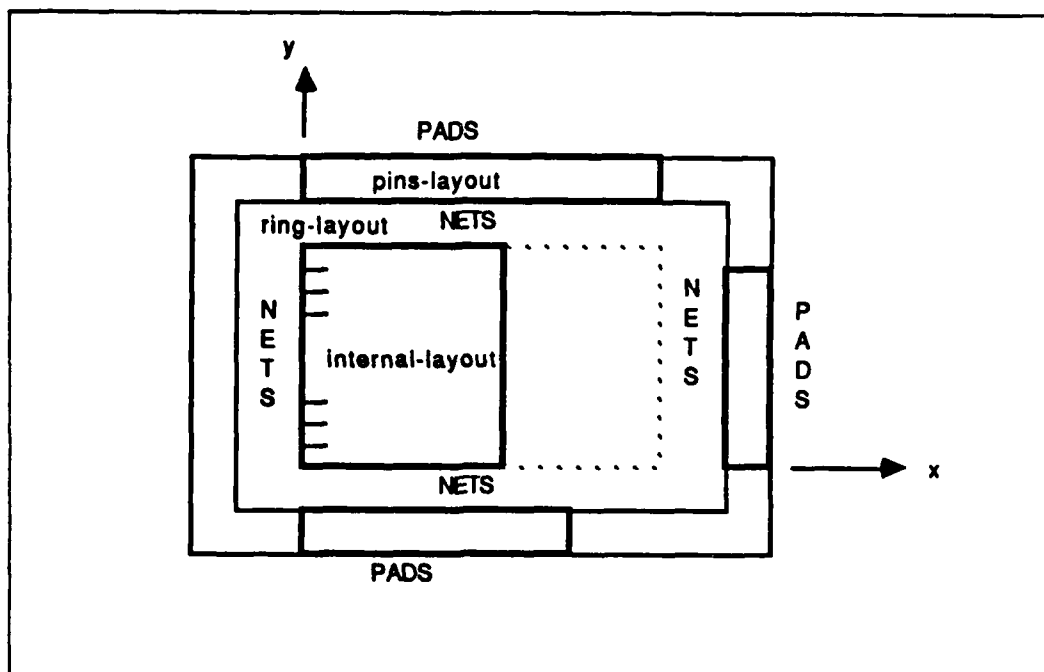


Figure 2.2 Ring-Layout Constraint.

and the length of the internal layout is called the **intended-right**. Given these values for length and width the pads are placed at an extended-top and extended-right position. The pads on top are aligned at $x = 0$, the right is at $y = 0$, the bottom is at $x = 0$ and the left is at $y = 0$. Based on a simple placement algorithm, the pads are placed clockwise around the circuit. The number of pads per side is found by dividing the total number of pads by the value 3. Rounding the result down to the nearest fixed number and incrementing by the value 1, gives the *#pins-per-side*. The first one-third of the pads are placed on the top side, the second third on the right side and the remainder on the bottom side. A division by the value 4 would allow the pads to be placed on the left edge of the circuit. However, a *namestack* overflow condition occurs because the ring layout algorithm is not designed to run wires from the left side of the chip. The net wiring in the ring channel requires decoding to allow net connections at the left edge of the chip. Since all wiring is directed to and from the internal-layout on the left edge of the circuit, long wiring runs around the ring and internal to the buses can be seen.

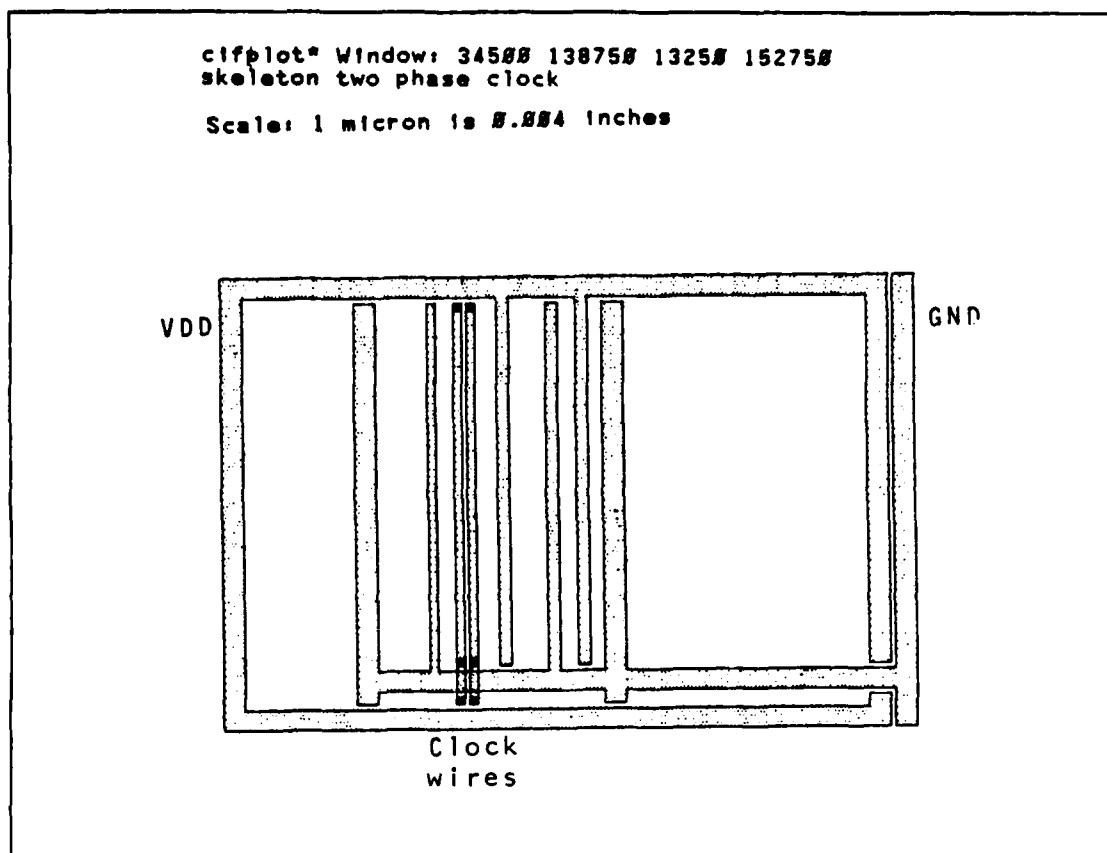


Figure 2.3 Skeleton Power/Ground/Clock Distribution Bus.

3. Internal Level of Design

The datapath is organized as the assembly of an array of units. A *unit* is n bits of either an *organelle*, a *register*, a *port-output*, a *port-internal* or a *bit wire*. An *organelle* is a bit-slice arithmetic/logic/shift cell that is maintained in the external library. The built-in organelles are the register cells, the ports and bit wire. These organelles are maintained in the data-path.l file. Units are connected by buses that run in the horizontal direction between bit-slice organelles/registers/ports/bit wires. Buses run on collinear tracks. The tracks are segmented so that the bus lines do not have to stretch to the ends of the datapath. The segments of the tracks are called internal-buses or local buses. They pass output signals from one unit to the input of another unit. The longest bus wire is that of the port-output and that depends on its position in the datapath. Positioning of units in the datapath is done by the ordering routines in order.l. Figure 2.5 illustrates the unit layout of a four unit datapath with the internal

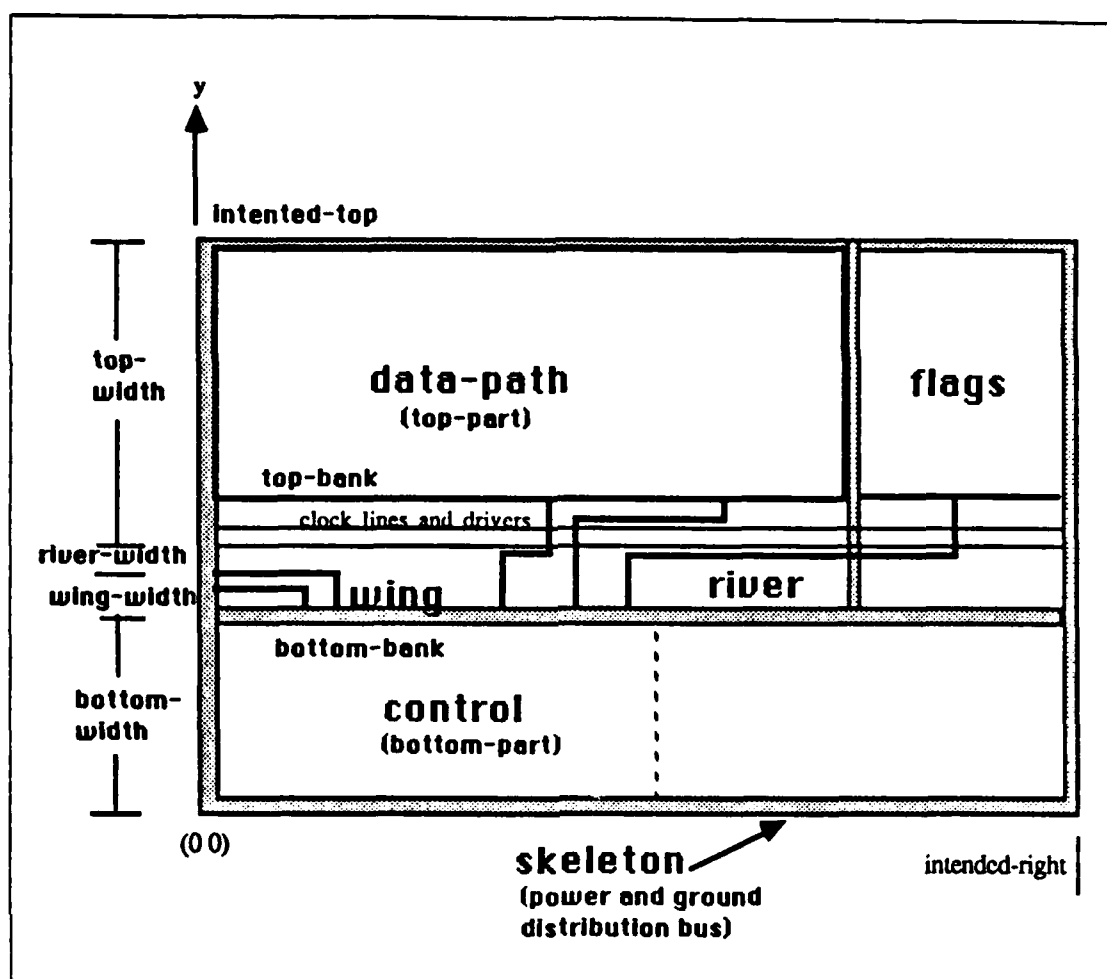


Figure 2.4 Internal Layout.

bus composed of two tracks. This figure also illustrates the architecture within a single unit.

In general, along a vertical slice of the datapath, organelles are stacked to form a unit of a length specified in the input program by the word-length. Each horizontal slice corresponds to a single bit slice of each unit in the datapath. The architecture of a bit slice unit is composed of an **organelle-part**, **river-part** and **mpx-part**. Mpx-part can be a multiplexer or single metal wires tapping off of the polysilicon bus lines. The multiplexer is connected to the inputs of the organelle by a diffusion wire generated by the river router.

Organelles are functional units interspersed between registers to implement the operations required by the input algorithm. Functional units such as adders,

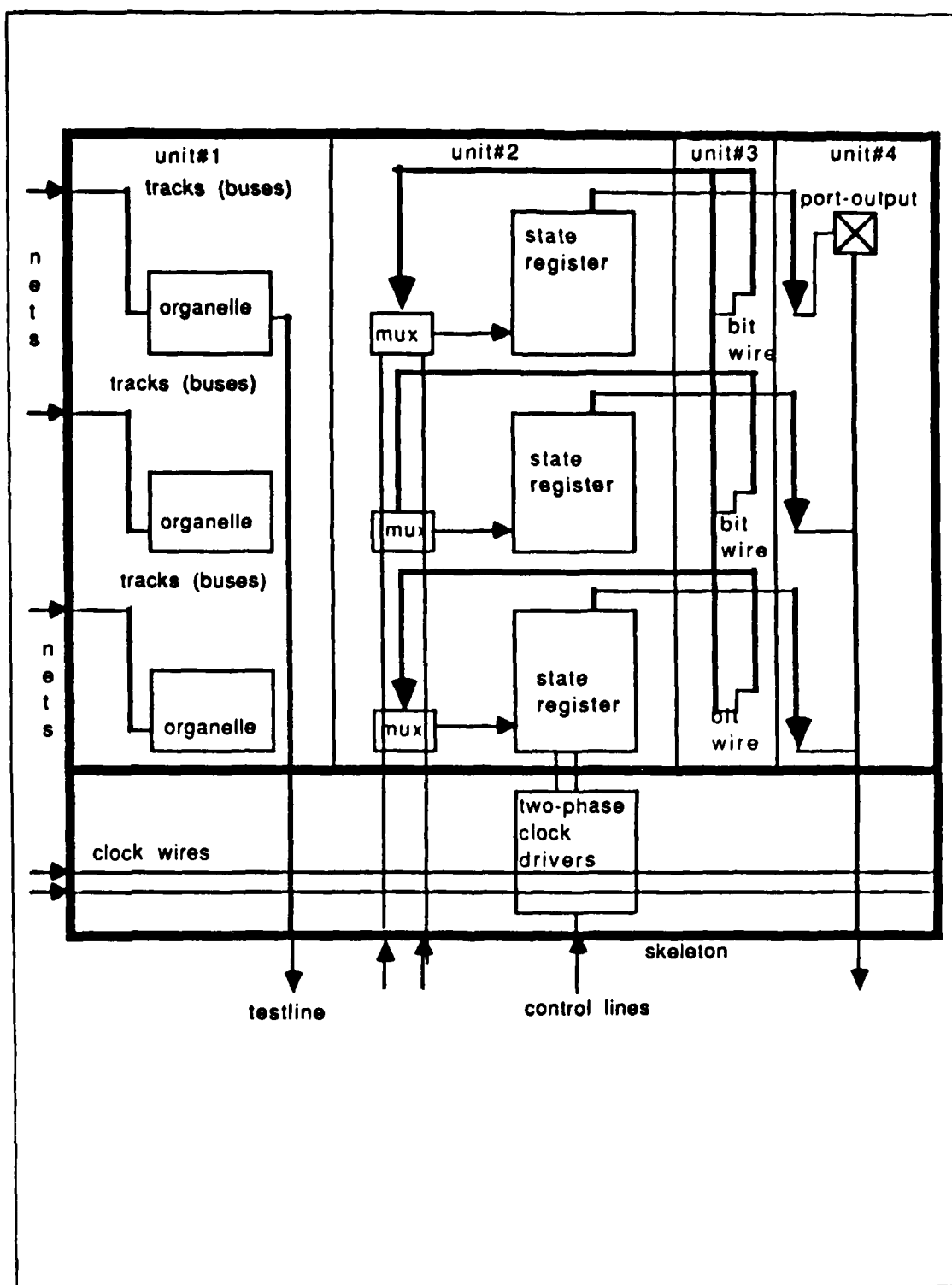


Figure 2.5 Unit Layout of a Datapath.

incrementers, logic cells identified in the input program as (word-nand, word-nor, word-xor) compute the values of forms used in **setq** forms in the input program. MacPitts has two data types: *integer* and *boolean*. Functional units produce integer outputs. That is, the output of a functional unit is a full word used by datapath. Test units, on the other hand, produce boolean outputs that are passed to the control unit. The library file contains a **function** construct that interfaces between the symbol used for an operation in the input program e.g., (word-and word-equ word-xor ...) and the name of that operator in the organelle data structure in the library, e.g., (and equ xor.....). The function construct is of the form (*function* <function-name> <organelle-name>). The organelle construct then calls the layout of that operator by the cell definition function (*layout*-<name>-organelle).

Test units such as the equality (=) or <> provide output test results to the control unit to implement primitive conditions used in the **cond** form. Test units do not pass their results to the datapath. When constructing the datapath MacPitts stretches the connection points of the organelle to connect with power and ground rails and with the data bus and test/control lines if required. Test comparator output lines are not stretched, but are daisy chained internally to the edge of the datapath for connection to the controller.

The layout of each organelle is generated by a LISP function (*layout-gen-form*), called by the compiler. The function passes an **instantiate** command to the organelle data structure in the library. Information on the dimensions of organelles, interconnection and power requirements, as well as the call to instantiate the layout of the cell, is contained in the (**gen-form**) field of the organelle data structure, the form of which is:

```
(organelle <name> <#control-lines> <#parameters> <#testlines> result?
      (gen-form)
      (sim-form))
```

This construct is a specific case of the *definition* desfstruct. The information contained in the **gen-form** function is illustrated in the functional block diagram of an organelle's bounding box and connection points in Figure 2.6. The **gen-form** function is defined in Table 1.

MacPitts uses this information to properly space the organelles and to stretch their connection points to connect with the control lines, power lines, and local interconnection buses. The insertion of SCMOS cells required that the information

TABLE 1
GEN-FORM

```
(lambda (info bit word-length drive ratio)
  ;The info argument is a field that matches any of the fields
  ;listed in the body of the gen-form.
  (cond
    ((eq info 'instantiate)(first-quadrant
      (layout- < name > -organelle drive ratio))))
    ;Call to layout the cell. First quadrant positions the origin at
    ;the lower left edge.
    ((eq info 'length) x)
    ((eq info 'width) y)
    ((eq info 'inputs) '(y1 y2))
    ((eq info 'output) '(x5))
    ((eq info 'vdd) '(x2))
    ((eq info 'gnd) '(x3))
    ((eq info 'daisy) '(x1 x4))
    ;Daisy is a form of pitch matching the output of a bit slice cell
    ;to the input of the next bit slice cell
    ((eq info 'output-type) '(ratio))
    ((eq info 'conductivity) (x))
    ;Conductivity values used to size power and ground buses for the
    ;internal layout.
    ((eq info '#transistors) '(n p))
    ;The value n is the total number of transistors and p is the
    ;total number of pullups in the organelle.
    ((eq info 'drive) '(x))
    ;In the NMOS version of MacPitts
    ;a pass transistor is appended to the output of a unit if the
    ;unit's bus number is positive. The pass transistor drives the
    ;gate of the unit to which it passes its output signal.
```

contained in the gen-form match the characteristics of the SCMOS organelle and that the layout geometry of the cell be placed in the compiled library, organelles.l.

4. Conversion Procedure

Converting from one technology to another requires an intermediate form in which both technologies may coexist. This form is known as **hybrid** because it is dual technology. The program under conversion to SCMOS technology is referred to as a hybrid program because the circuits produced by the compiler contain a mix of NMOS and SCMOS layouts.

The procedure for inserting an SCMOS organelle into MacPitts begins with the conversion of the organelles' CIF code to rectangular code and formatting the code so that it is parenthesized with a header of (*defsymbol <name> nil*). This form will be explained in more detail in Chapter IV section B. A simple conversion routine was written converting the CIF code to rectangular code. Cif geometry is formatted in the box form:

(B length width xcenter ycenter)

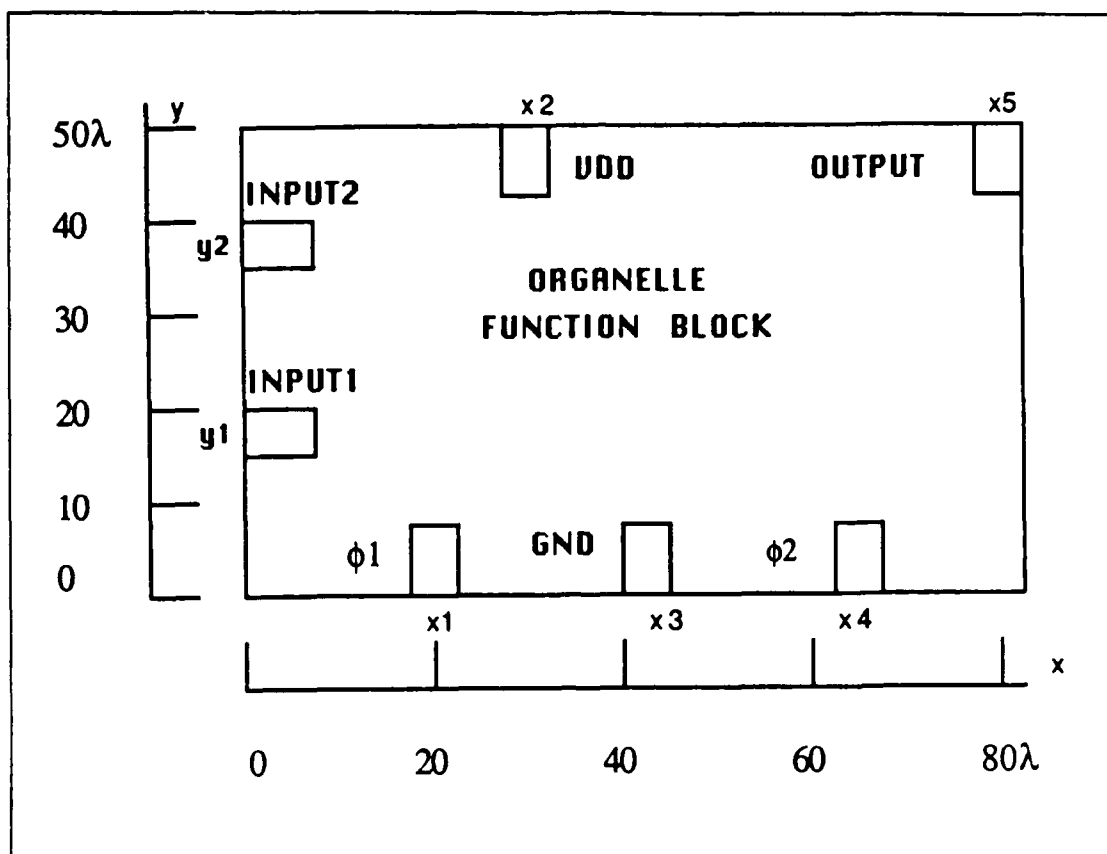


Figure 2.6 Organelle Functional Block.

whereas, L5 form is formatted in rectangular form:

```
(defsymbol < name > nil (merge (rect xmin ymin xmax ymax)))
```

A cell definition function is created of the form *(layout-< name >-organelle)*. The function calls the L5 code or defsymbol containing the geometry of the cell. The attributes of the organelle contained in the gen-form field of the organelle data structure in the library is altered to reflect the size and interconnection points of the SCMOS cells. The cell definition and the geometric layout functions are loaded into the macpitts environment and plotted for dimensional analysis using the functions *(thesis-plot)* or *(plot)*² in the file plot.l. The user must be in the LISP or macpitts environment when using these functions. The organelle is plotted and its dimensions

²The format of the thesis-plot function is:

```
(thesis-plot (layout-< name >-organelle drive ratio)
```

title external micron lambda) is the form of the thesis-plot function. An example is: `-> (thesis-plot (layout-xor-organelle 0 '(4 4)) 'xor t 1.5)`

manually measured from the stipple plot using the simple conversion of ((scale / feature-size) * measured value). ³ The locations of the interconnections and the length and width are entered into the gen-form of the particular organelle being changed. The gen-form function is loaded at run time into the macpitts environment. An input algorithm is written in the form of a <filename.mac> program to test the organelle for misalignments. For example, to generate an inverter circuit, the .mac program is written as in Table 2. To test the various library functions, this same program is run with the names of the functional library forms: word-and, word-nand, word-xor word-nor, word-or, word-equ.

TABLE 2
MACPITTS INPUT PROGRAM

```

(program < name> < word-length>
(program macnot 1
(def < pin-number> ground)
(def 1 ground)
(def < pin-number> clock)
(def 2 phia)
(def 3 phib)
(def < pin-number> power)
(def 6 power)
(def < name> port <input output tri-state i/o>
(( < pin-numbers> *) internal)
(def a port input (4))
(def b port output (5))
(always [ < form> ]*)
(always
  (setq b (word-not a)))
  (setq < port-name> < form> )
  < form> = (word-not integer)
  Returns the complement of the integer. Word-not is a library
  function

```

The *setq* form in this program causes the datapath to evaluate a sequence of operations on input data, in this case, data into port a. The result is output from the circuit through a port, in this case, port b at the end of the clock cycle. The *always* construct results in a stateless process whereby its functions are executed every clock cycle. This program is written to a file in the VI editor on the UNIX system with a .mac extension. While in the macpitts environment, the command (macpitts

³Scale is read from the header information on the stipple plot. Feature size is 1.5 microns per lambda for SCMOS or 2.5 microns per lambda for NMOS.

<filename>) is executed. The execution returns a <filename>.cif and <filename>.obj.

The .mac input program is read concurrently with the data file, *library*, during compilation. The compiler's higher level routines examine this source code and extract a technology independent intermediate level description of the system in the object file, which is given for the inverter example in Table 3.

TABLE 3
OBJECT FILE

```
object file
definitions list
(destination b)
(source a)
(logic macnot)
(ground 1)
(port a input (4))
(port b output (5))
(phia 2)
(phib 3)
(power 6)
flags specification
nil
datapath specification
((organelle not -1 (((port-input a))))
(port-output b(((internal 1))))))
control specification
nil
pins specification
((3 phib))
((2 phia))
((1 ground))
((6 power))
((4 input a 0)(port-input a 0))
((5 (output8 (b 0) (port-output b 0))))
```

Each of these forms is specified by the defstruct *definition*. The circuit layout is then specified in the .cif file produced by the compilation process. Upon completion of the execution of the program, the cell is plotted and checked for misalignments.

To plot dual layer hybrid circuits, the file **patterns** was created containing stipple designs for NMOS and SC MOS layers. The cifplot command is invoked to generate stipple plots of circuit designs.

cifplot [options] <filename> .cif

In order to work in a dual NMOS and SCMOS environment in MacPitts the (*allowed-layers*) function was modified to accept both NMOS and SCMOS layers creating a HYBRID environment. ⁴ Additionally, the (*macpitts-compiler*) function was modified to accept 3 micron minimum-feature-size. The function (*pad-class*) was altered to allow plotting of NMOS pads while invoking the 3 micron feature size. The pads located in the file *rinout* are plotted at a minimum-feature-size of 250 centimicrons per lambda. The pads located in the file *pad20b* are plotted at a minimum-feature-size of 200 centimicrons per lambda. The SCMOS pads have not been inserted yet. [Ref. 3] discusses these modifications in more detail.

5. Key Constructs

The key ideas relevant to the porting of a new technology into MacPitts is the ability to build layout structures from list structures known as *item* and to make *symbols* out of lower level units, place them in a symbol table, and call the symbol-number as many times as required. Each call to the symbol number places a copy of that cell with the origin at a designated position on the grid of the cell doing the calling. An item is a structured list that contains information on a cell, its bounding box, its points, calls to other cells used to hierarchically build itself and its rectangular geometry. It is a long defstruct of the following form:

(item left bottom right top points called-symbol-names tree)

The layout geometry of a cell is contained in the symbol-making construct *defsymbols* ⁵ as defined in the program L5.1.

When a call is placed to that *defsymbols*, the macro checks to see if the cell has a symbol number on the symbol table, -L5-symbol-list. If it does not, it converts the geometric code into an item form, and extracts information from the item to build a symbol and place that symbol on the -L5-symbol list. It then converts the rectangular code into CIF and places it in the -L5-symbol-file. Symbol is a defstruct of the form:

(symbol ID left bottom right top points internal-symbols nest-level tree)).

Defsymbols are a space and time saving device. However, as technology changes, and cells such as pads are proportionately larger because of shrinking feature sizes, the conversion of large amounts of CIF code to rectangular code become unfeasible. The recursive powers of LISP were not designed to hold 940 lines of CIF code on a stack while processing the conversion. A new symbol making structure that addresses this

⁴An in-depth discussion of SCMOS layers is contained in the file "text" in /vlsi/berk83/doc/magic/scmos.

⁵Defsymbols is a LISP macro. Macros produce code and then evaluate the code.

difficulty by extracting symbol information directly from the CIF code is currently under development. [Ref. 3]

Defsymbols and items allow for the hierarchical construction of the circuit. Hierarchically constructed modules such as the datapath, control, flags, pins, skeleton, and wing are defined as defsymbols. Small item structures grow into big item structures with calls to the lower level units. An illustration of an item is given in Figure 2.7. The top-level layout is a set of calls to lower level cells. Each call to a cell places a copy of that cell with the origin at a designated position in the grid of the cell doing the calling. Figure 2.8 diagrams the hierarchy of calls for a MacPitts circuit composed of an equality (=) organelle and a register.

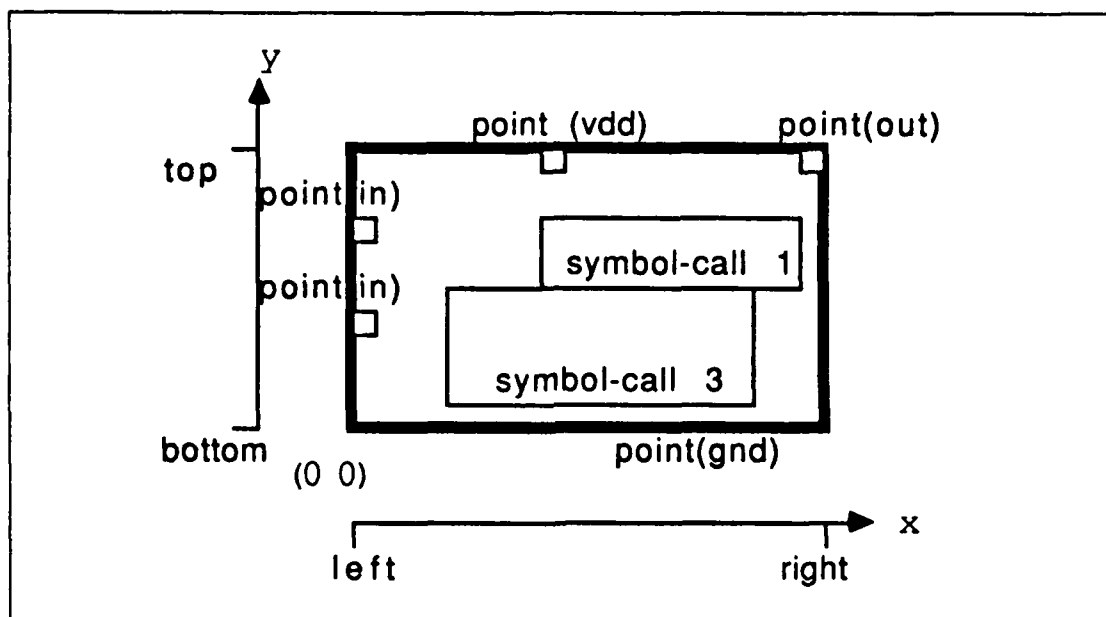


Figure 2.7 Item Topology.

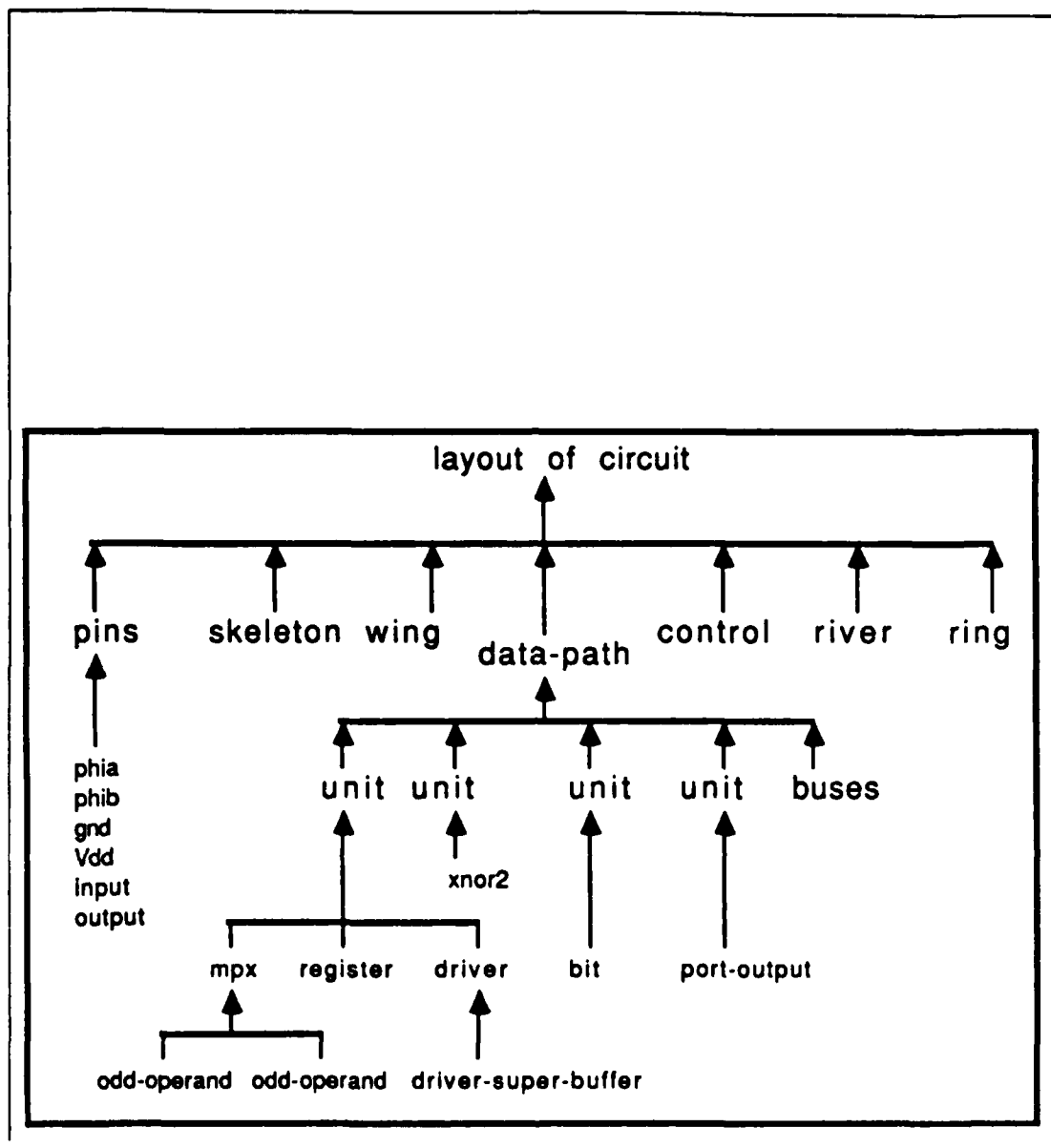


Figure 2.8 Hierarchical Construction of a MacPitts Circuit.

III. DATA-PATH LAYOUT

A. METHODOLOGY

The method of committing the datapath specification extracted from the input program (.mac) and located in the object file (.obj), to an L5 geometric layout in the form of an *item* is the subject of this chapter. Specifically, the datapath generated for a four bit **HYBRID nand circuit** containing an SCMOS nand organelle in an NMOS circuit layout is examined.

This input program **nand.mac**, is a behavioral description of the 4 bit **nand** circuit. Table 4 lists the input program. The program is written to a file with a .mac extension. For example, the above program is written to a file named **nand.mac**. The program is executed by invoking the **macpitts** executable command on the UNIX command line.

macpitts < filename >

where the filename in our example is 'nand' without the .mac extension. Statistics are printed to the terminal screen on each execution and may placed in a file, e.g., **< filename > .stat** as follows:

%macpitts nand > nand.stat &

The statistics of the circuit generated are placed in the file **nand.stat**. The **&** places the process in the background. The technology independent intermediate description of the circuit in **nand.obj** is generated. The object file is listed in Table 5.

The object (.obj) file is a list containing five sublists: *definitions flags*, *datapath*, *control* and *pins*. The *datapath* sublist contains the technology independent description of the structural composition of the *datapath* that is then translated into a layout. The *datapath* is best described as an array of units. Each list within the *datapath* list describes a structure of each *unit* in the datapath array. The *datapath* list is called a *unit-list* for this reason. The *unit-list* for the **HYBRID nand** circuit is composed of two units: *organelle* and *port-output*.

Each unit in the *unit-list* has its own circuit structure. A unit may be a *register* cell, *organelle*, *port-output*, *port-internal* or a *bit*. These five circuit types are the building blocks of the datapath. An *organelle* can be an integer or boolean logic organelle. Integer logic organelles are (*word-nand*, *word-and*) and boolean logic organelles are (*nand*, *nor*, *not*, *and*, *or*, *xor*, *equ*). Other types of organelles include an

TABLE 4
INPUT PROGRAM FOR A 1-BIT NAND CIRCUIT

(program nandchip 1

;The circuit is a 1 bit nand gate with two 1 bit input data buses,
;one 1 bit output data bus, a two phase clock bus and a ground and
;power bus. Clock phases must be defined in all MacPitts programs
;whether the circuit is clocked or not.

(def 1 ground)
(def a port-input (2))
(def b port input (3))
(def c port output (4))
(def 5 phia)
(def 6 phib)
(def 7 power)

;The LISP form (word-nand a b) takes the logic nand of two inputs and
;places the result on the output bus (port c). This is done every clock
;period as specified by the 'always' construct.

(always
 (setq c (word-nand a b))))

arithmetic organelle (+ - / + /-), a test comparator organelle (= < > < > 0 ..) and left or right shift organelles: (< < 2 < < 3 < < 4 < < 8 < < > > 2 > > 3 > > 4 > > 8 > >). Organelles are contained in a cell library. The organelle layout information is contained in the compiled source program, *organelles.l*. The uncompiled file, *library* contains an *organelle data structure* that contains a file of information on each organelle. The type of information found in this data structure is the *length* and *width* of the organelle, i.e., its bounding box, the location of its I/O, Vdd, and gnd terminal points along the edges of the bounding box, the number of transistors in the organelle as identified in the gen-form as *#transistors* and the power requirements as identified by the field *conductivity*. The portion of the organelle data-structure that contains this information is called *gen-form*.

1. Defstructs

These concepts are implemented by the *defstruct* data structure. To retrieve data by name, a construct is created from LISP called a *defstruct*. For example, if A is an array, A[i] is a name for one of its elements. In the assignment B := A[i], the name B refers to the value of A[i]. In applicative languages like LISP, an array is defined as a list, e.g., (setq A '((A1 A2)(B1 B2))). The value of a field in an array is found by progressive search of the head or tail of the list. LISP uses the nonmnemonic

TABLE 5
OBJECT FILE FOR A NAND CIRCUIT

```

Definitions
  (((destination c)
    (source a)
    (source b)
    (logo macnand)
    (word-length 1)
    (ground 1)
    (port a input (2))
    (port b input (3))
    (port c output (4))
    (phia 5)
    (phib 6)
    (power 7))
Flags Specification
  nil
Datapath Specification
  ((organelle nand -1 (((port-input a)(port-input b))))
    (port-output c (((internal 1)))))
Control Specification
  nil
Pins Specification
  ((6 (phib))
    (5 (phia))
    (1 (ground))
    (7 (power))
    (2 (input (a 0)(port-input a 0)))
    (3 (input (b 0)(port-input b 0)))
    (4 (output8 (c 0) (port-output c 0))))

```

names CAR and CDR for the search for the head or tail of a list. To find the value of A1, the Lisp function (*caar 'A*) returns the value A1. To retrieve the same data by name, a long defstruct of type 'array' is created. The user enters the *macpitts* environment and types the following:

```

(defstruct array
  row1 (first second)
  row2 (first second))

```

The array is created by the (*make-type field*) creator function.

```

-> (setq row1 (make-row1-array '2 3) <cr>
   (row1 2 3))
-> (setq row2 (make-row2-array '5 6) <cr>
   (row2 5 6))

```

To select an array value:

```

-> (row1-array-first row1) <cr>

```

To change a value

```
-> (replace-row1-array-second row1 8) <cr>
(row1 2 8)
```

A short defstruct for the array example provides another result.

```
(defstruct array1
  (row1 row2))
-> (setq array '((2 3)(5 6))) <cr>
((2 3)(5 6))
-> (make-array array) <cr>
(((2 3)(5 6)))
```

To select a value:

```
-> (array-row1 array)
(2 3)
```

To replace a value:

```
(replace-array-row2 array '(4 5))
((2 3)(4 5))
```

The defstruct structure is an automatic CDR generator and is defined in the source program *lincoln.l*. The short defstruct now exists in the basic Franz Lisp symbol table as a primitive. The long defstruct has not yet been incorporated into Franz Lisp.

The following defstructs define the structure of the datapath.

```
(defstruct data-path
  ((unit))
```

The unit defstruct is defined as:

```
(defstruct unit
  organelle (name bus# mpx)
    ;;(organelle + -1 (((internal 8) (constant 10))))
  register (name bus# mpx)
    ;;(register timer -2 (((constant 0)(internal 4))))
  port-output (name bus# mpx)
    ;; (port-output display (((internal 7))))
  port-internal (name bus# mpx)
    ;;(port-internal carry -3 (((constant 1))((constant 0))))
  bit (bit-list mpx) )
    ;;(bit (0) (((internal 5))))
```

The *mpx* is a defstruct composed of an *argument*. An *argument* is composed of an *operand*. The *operand* types are (*port-input name*), (*internal bus#*), (*constant number*) and so forth. *Mpx*, *argument*, and *operand* are defstructs defined in the source program *defstructs.l*. Understanding the nesting among the defstructs makes the datapath specification in the object file comprehensible. The datapath specification in the .obj file for the HYBRID nand circuit is composed of a unit-list of two elements:

```
unit 1 <- (organelle nand -1 (((port-input a)(port-input b))))
          ;(organelle name bus# mpx)
          ; mpx <- (((port-input a)(port-input b)))
unit 2 <- (port-output c ((internal 1)))
          ;(port-output name mpx)
          ;mpx <- ((internal unit#))
```

The form (*internal 1*) indicates an internal bus from unit#1 connecting to unit#2.

B. ARCHITECTURAL IMPLEMENTATION OF A UNIT

The layout of an *organelle*, *register*, *port* or *bit* is accompanied by the layout of either a multiplexer or single vertical wires. The organization of a *unit* is a (*mpx-part*), (*organelle-part*), and a channel (*river-part*) where the river-router connects the *mpx-part* to the *organelle-part*. The *mpx-part* is a generic term for the placement of a metal wire or multiplexer at the inputs of an *organelle*, *register*, *port* or *bit*. The data buses lie in the horizontal channel between bit slices of organelles. *Tracks* are the number of horizontal bus lines. These lines are segmented on the same collinear track allowing units to connect to each other on short local or internal bus lines. The bit slice unit is stacked along a vertical slice of the datapath to form a unit.

1. Database

The datapath floorplan parallels the organization of the database, i.e., the LISP functions that create the layout of the data-path. The top-level function (*layout-data-path*), coordinates the layout of the data-path. It receives the *unit-list* or datapath specification from the object file as an argument, and returns the layout of the datapath in *item* format.

datapath layout <- (item left bottom right top points called-symbol-names tree)

The *unit-list* is recursively processed by the function (*layout-unit-list*). The individual *unit* is processed by the function (*layout-unit*).

A unit-type can be either an *organelle*, *register*, *port*, or *bit* of n-bits as specified by the *word-length* in the .mac program. Each unit type is recursively

processed by the function (*layout-organelle-list*). The actual layout of a unit type is accomplished by the function (*layout-organelle*). The multiplexer is generated by the (*layout-mpx*) function. The unit-type is generated by the (*layout-gen-form*) function. The (*river*) function generates the wires that connect the two structures. Figure 3.1 provides a flowchart of the functions that process the datapath specification into a layout. Figure 3.2 illustrates the topology of the datapath and the LISP functions that generate them.

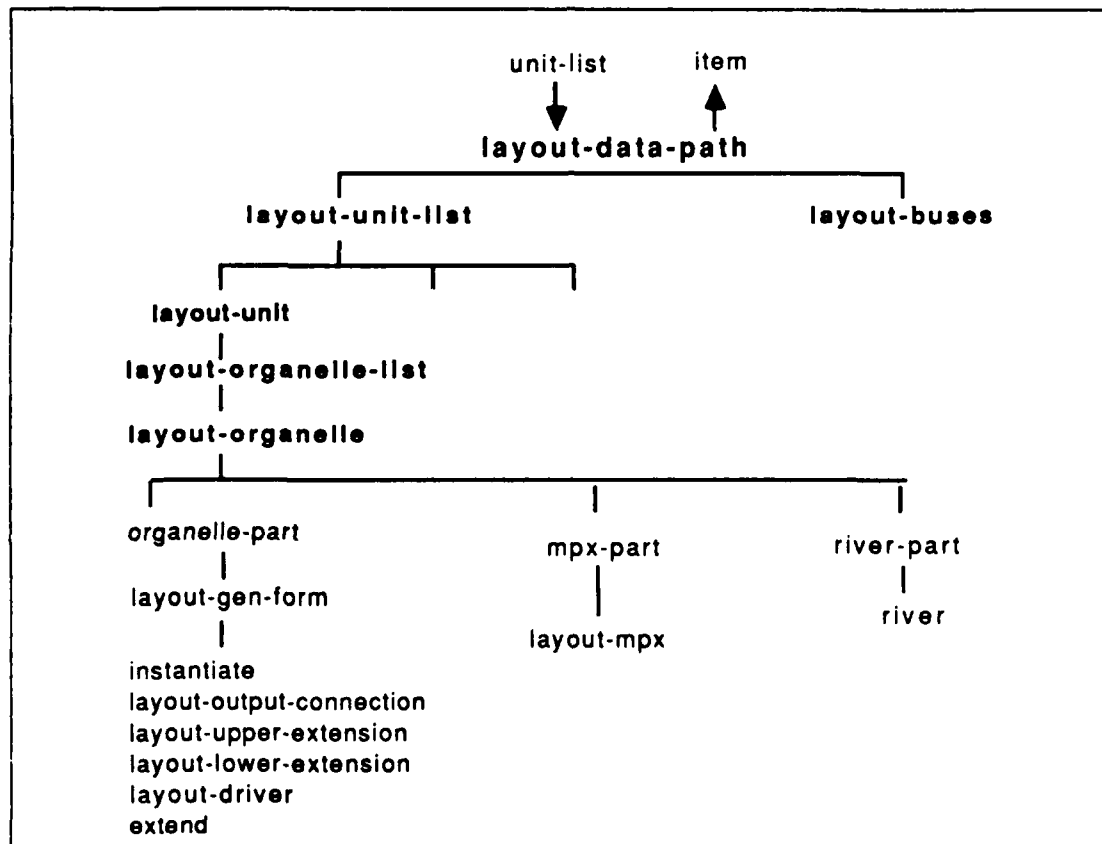


Figure 3.1 Flowchart of Functions in Datapath.

C. ORGANELLE-LIST

1. Data-Path Placement Algorithm

The *datapath* placement algorithm begins with the function (*layout-organelle-list*). For an *n*-bit word organelle, the most significant bit (MSB) is positioned at $(x\ y) = (0\ 10)\ \text{lambda}$. The rest of the bits are placed relative to the MSB at

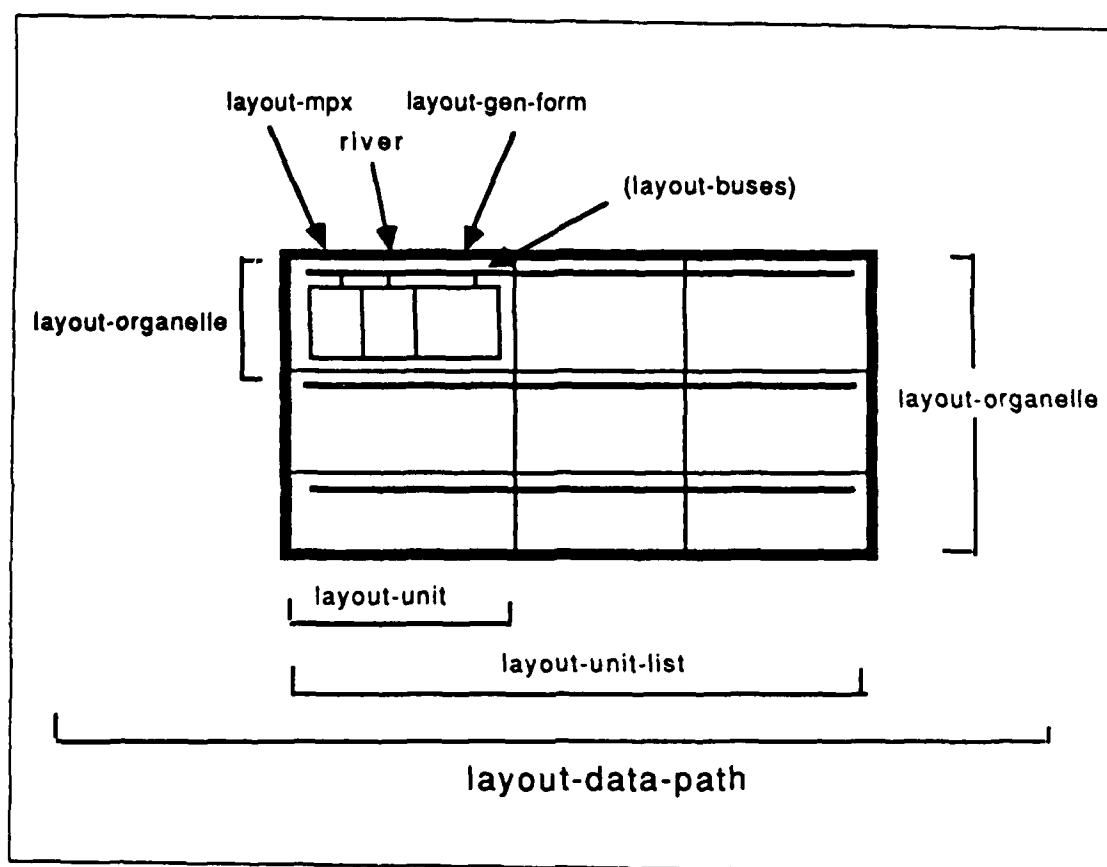


Figure 3.2 Functional Layout of the Datapath.

$$x = 0 \text{ and } y = y + 4 + (5 * \text{tracks}) + \text{organelle-width}.$$

The channel width of $(5 * \text{tracks})$ in this equation accommodates the buses. This factor is based on a wire width of 3 lambda with a spacing of 2 lambda between tracks. Figure 3.3 illustrates the placement of the organelles within a unit. A unit's width is defined by the *maximum-width-list*. This list contains the width of the largest operator in the datapath repeated by the number of bits in the *word-length*.

2. Layout Organelle

The (*layout-organelle*) function coordinates the generation of the bit-slice unit. The bit-slice of a unit requires the layout of the mpx-part, organelle-part and river-part. It receives a unit-type (*organelle*, *register*, *port-input*, *port-internal*, *bit*), as an argument and calls the (*lookup-gen-form*) function to get the unit's *gen-form*.

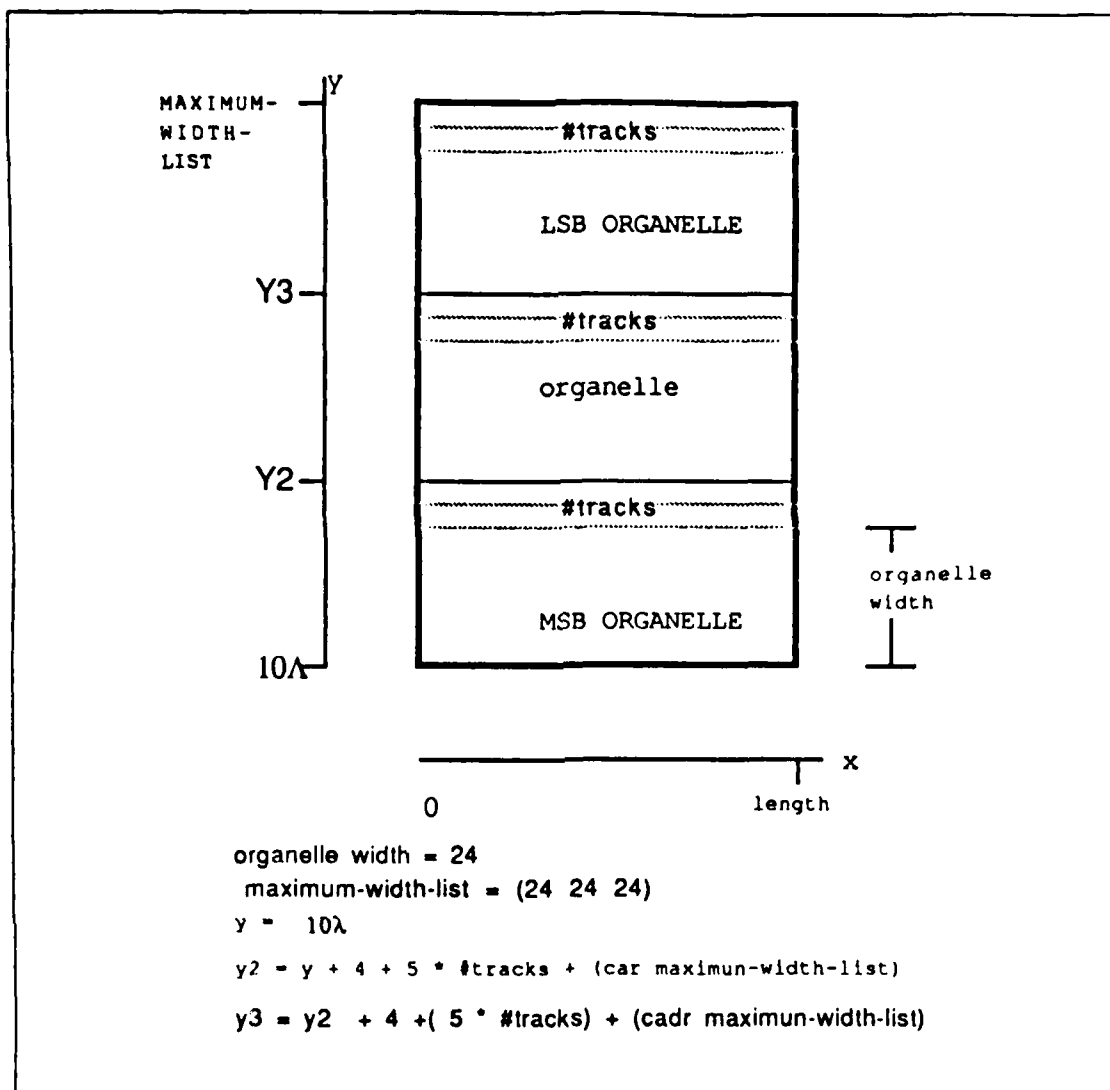


Figure 3.3 Unit Placement.

a. Gen-Form

Gen-form is a function within the *organelle* data structure in the *library* and within the layout functions for the *register*, *port-output* and *bit* in the source program *data-path.l* that gives pertinent information on the unit-types dimensions, terminal connections and power requirements. The function (*lookup-gen-form*) determines if the *unit* is a *register*, *organelle*, *port* or *bit*. If the unit is an *organelle*, the *gen-form* of the *organelle* in the *organelle* data structure is looked up in the *library* by the function (*lookup-gen-form1*). The defstruct selector function (*organelle-definition-gen-form*)

returns the *gen-form* of the organelle. The LISP code below assigns the keyword *gen-form* to the values of the *gen-form* function.

```
(let ((gen-form (lookup-gen-form unit definitions)))
```

If the *unit* is a *port-output*, or *port-internal*, the (*layout-port-output*) function is returned. If the unit is a *bit*, the (*layout-bit*) function is returned. If the unit is a *register*, the (*layout-register*) function is returned. These layout functions contain their own *gen-form* function. The SC MOS nand *organelle gen-form* is given in Table 6, and a functional block diagram illustrating the locations of the terminals listed in the *gen-form* is shown in Figure 3.4.

3. Tail

MacPitts provides constructs to allow the datapath layout functions to query the values of the attributes of instantiated organelles. The arguments passed to the *gen-form* of an organelle are determined by the (*get-tail*) function. A *unit* that is either a *register*, *port*, *bit*, or *organelle* is passed as an argument to the (*get-tail*) function and a list of values for *bit#*, *word-length*, *drive* and *ratio* is returned to the (*layout-organelle*) function. The lists returned by the (*get-tail*) function depend on the unit-type passed in as an argument. Table 7 provides the list returned for each unit-type.

For example, the 1 bit SC MOS nand *organelle* has the following *tail* assigned to it.

```
tail <-- (0 1 -1 (4 4)).  
         (bit# #bits drive ratio)
```

These values match the arguments passed to the *gen-form* of the nand *organelle* in Table 6. The *organelle's gen-form* has the arguments (*info bit word-length drive ratio*). The values of the arguments are *bit* = 0, *word-length* = 1, *drive* = -1 and *ratio* = (4 4). The *drive* value is the *bus#* of the *organelle*. If the value of the *bus#* is > 0, a passgate is appended to the output of the NMOS *organelle* and is gated by a control signal, otherwise the basic NMOS *organelle* is instantiated. The *ratio* is used to size the pull-down transistor gates at the inputs of the nand *organelle*. The *info* form is any one of the fields in the *gen-form* function (*instantiate*, *width* *length*, *inputs*, *output*, *vdd*, *gnd*, *drive*, *daisy*, *test*, *#transistors* *conductivity*)

a. Ratio

The *ratio* of an *organelle* is required in the NMOS version of MacPitts to hierarchically construct the *organelle*. The *ratio* for each unit in the data-path is determined by the function (*data-path-ratio*). The (*unit-ratio*) function determines the *ratio* of each unit. The HYBRID nand circuit is composed of two units: a SC MOS

TABLE 6
SCMOS NAND ORGANELLE GEN-FORM

gen-form ::= (lambda (info bit word-length drive ratio)
(cond

if info = instantiate, then (eq info 'instantiate) returns a 't' value causing the execution of the (layout-nand-organelle ...) function. The function (first-quadrant) places the lower left corner of the organelle at the origin.

((eq info 'instantiate)
(first-quadrant(layout-nand-organelle drive ratio))))

The measured length and width of the organelle is in lambda units. The orientation of the organelle is: inputs on the left, outputs on top.

((eq info 'length) 32)
((eq info 'width) 37)

Inputs are measured along the y-axis

((eq info 'inputs) '(16 24))

Daisy is measured along the x-axis. The SCMOS nand organelle has no outputs or clock requiring daisy-chaining.

((eq info 'daisy) ())

Output-type = ratio is for NMOS organelles. The SCMOS organelles do not have special output-types.

((eq info 'output-type) '(ratio))

Drive provides the location of the passgate appended to the output of of an NMOS organelle. The passgate is enabled by a control line. The SCMOS organelles are designed with built-in drive and do not require a drive structure.

((eq info 'drive) ())

Conductivity information is heuristically arrived at for static NMOS organelles. A heuristic value for SCMOS organelles is also required

((eq info 'conductivity) (quotient 1 50))

Power and ground locations are measured along the x-axis. These lines run through the vertical length of the unit.

((eq info 'vdd) '(18))
((eq info 'gnd) '(9))

Output is located on the top of the cell and is measured along the x-axis

((eq info 'output) '(30))

The #transistors is composed of a list of two fields. The first field is the number of FETS and the second field is the number of pullups. The function (data-path-transistor-count) determines the number of transistors used in the layout of the datapath by querying the gen-form of the organelles. The total number of transistors used by a MacPitts circuit is output as a statistic. The SCMOS nand cell has four FETS, two Nfets and two Pfets. The number of pullups are irrelevant for static CMOS.

((eq info #transistors) '(4 ()))

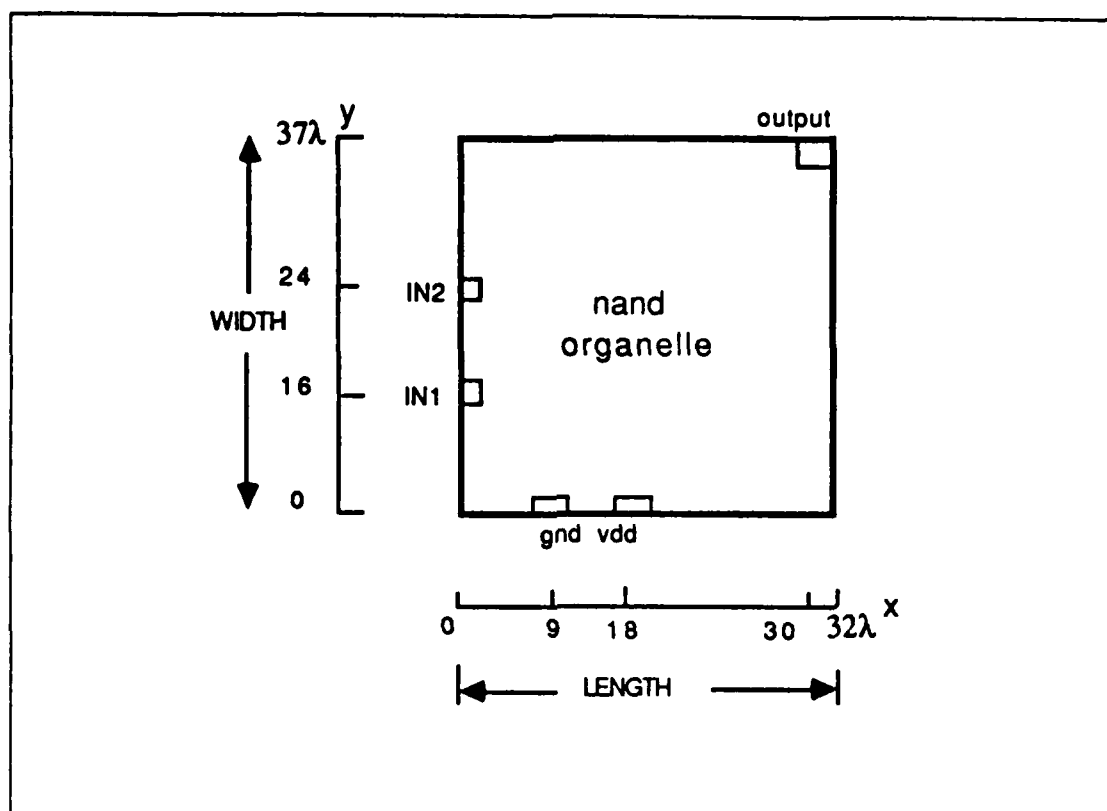


Figure 3.4 Bounding Box for a Nand Organelle.

nand organelle and a *port-output*. The datapath ratios for this HYBRID circuit is the list

data-path-ratios = ((4 4) 4).

The first list (4 4) gives the ratio for each pulldown transistor for each input of a NMOS organelle. The NMOS nand organelle is constructed from a pullup of fixed gate size (2 lambda x 8 lambda) and two pulldown transistors added in series to provide a two input nand gate. The pullup is kept to minimum-size. Overall ratio may be changed from 8:1 to 4:1 by adjusting, for example, the L:W ratio of the pull-down transistor. A 4:1 pulldown transistor has a 2 lambda x 2 lambda gate. An 8:1 ratio pulldown transistor has a 2 lambda x 4 lambda gate. The second unit, port-output, has a ratio of 4 as defined by the second value in the (*data-path-ratio*) list

SCMOS organelles do not use ratio as a parameter for cell construction, primarily because the cells are not hierarchically constructed by the MacPitts program *organelles.l*, but are completely defined at the gate level. Ratio is not used as a

TABLE 7
ARGUMENTS TO THE GEN-FORM

UNIT-TYPE	TAIL
register	(bit# #bits (register-unit-bus# unit) ratio) (bit# #bits drive-type ratio)
port-output	(bit# #bits 0 ratio)
port-internal	(bit# #bits 0 ratio)
bit	(bit# (bit-unit-list unit) ratio) (bit# bits-needed ratio)
organelle	(bit# #bits (organelle-unit-bus# unit) ratio) (bit# word-length drive ratio)

parameter in the generation of the SC MOS organelles, but is retained while working with a HYBRID chip.

4. Drive

A *port* is specified in the input program (.mac) as follows:

```
(def <port-name> port [input / output / tri-state/ i/o
  ([< pin-numbers]*) / internal])
```

The function (*extract-port-setq*) extracts *port* information from the behavioral specification of the chip in the .mac program. The function (*does-port-need-drive*) reads the port specification in the input program. If a *tri-state* or *I/O port* is specified, a passgate is appended to the output of the unit fed by that bus. Drive is indicated by the *bus#* of the unit type. A positive (> 0) causes the instantiation of the passgate to the output of the unit. The NMOS organelles, except the equality (=) and <> 0 cells, may have a passgate at their outputs if specified by the object file. Equality (=) and not equal (<>) organelles have a *bus#* = 0 to indicate that they do not pass their outputs to the datapath.

Bus numbers are assigned in the extraction of the datapath by the function (*assign-bus-numbers*) in sequential order. The (*post-process-bus#*) function transforms the *bus#* of each unit to a negative value (-1 -2 -3). Positive organelle bus numbers indicate the instantiation of a passgate at the output of the organelle.

SCMOS organelles were designed with built-in drive values. In particular, the three inverters with increasing drive capability are identified by the 1x, 4x and 8x extensions in the Mullarky SCMOS Cell Library. The remainder of the SCMOS library cells are assumed to have the minimum drive of 1x. In the SCMOS version of MacPitts, the need for additional drive to the SCMOS organelles, as a function of the use of I/O pads requires further investigation.

D. ORGANELLE LAYOUT

Figure 3.5 illustrates the topology of the bit-slice unit. The *organelle-part* is created by the (*layout-gen-form*) function. This function instantiates either an *organelle*, *register*, *port*, or *bit* and adds metal wire extensions to Vdd, gnd and output. The instantiation of the organelle is initiated by inserting the keyword *instantiate* into the list called *tail* and passing the *tail* to the *gen-form* of the organelle. The LISP code

```
-> (call-list gen-form (cons 'instantiate tail))
```

passes the *instantiate* command to the *gen-form* of the organelle causing the layout of the organelle. An *item* is returned containing the layout information on the organelle. The flow of execution in the Lisp environment is as follows:

```
tail      = (instantiate 0 4 -1 (4 4))
gen-form  ::= (lookup-gen-form unit definitions)
gen-form  = (lambda (info bit word-length drive ratio)
              (cond ((eq info 'instantiate....)

info = instantiate
;The equality is satisfied and a 't' value is returned.
;Layout the organelle and return an item containing the layout.
(first-quadrant (layout-nand-organelle drive ratio)
; drive = -1
; ratio = (4 4)
;An item is returned of the form
;item <- (left bottom right top (points) (called-symbol-names) tree)
```

The Macpitts function (*first-quadrant*) places the origin of the cell at the lower left corner. The (*layout-nand-organelle*) cell definition is located in the source program *organelles.l*. It calls the defsymbol containing the layout geometry of the cell. The call to the defsymbol causes the creation of an item out of the layout geometry.

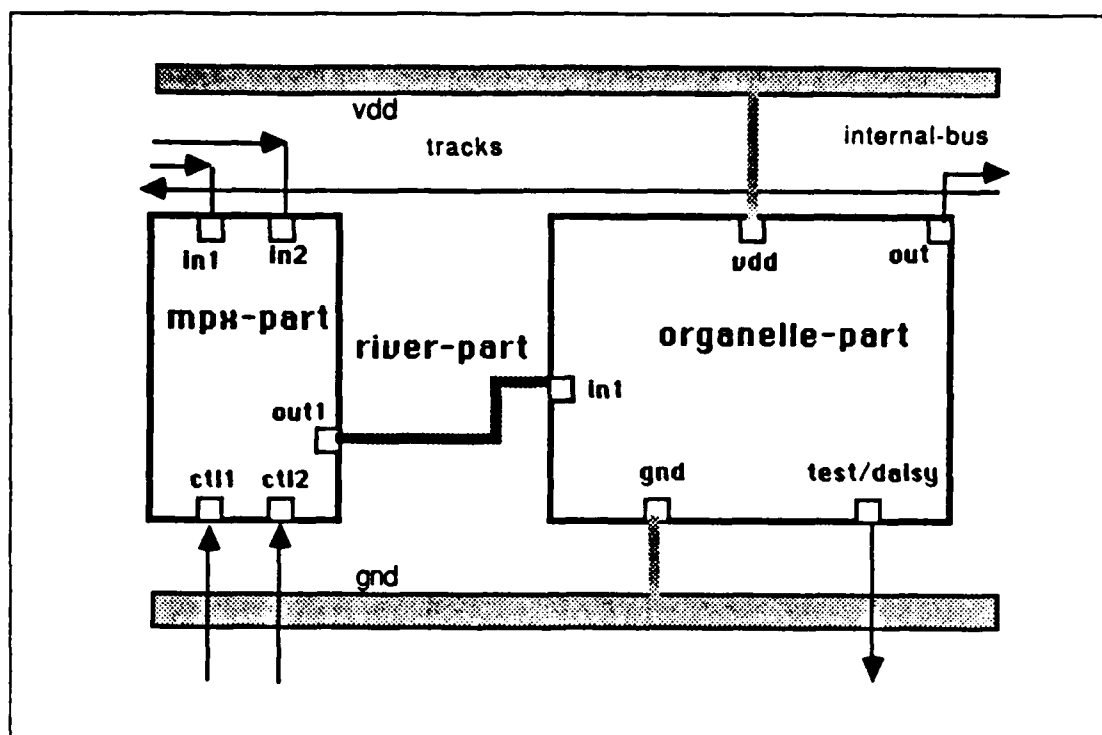


Figure 3.5 Bit-slice Unit Topology.

1. Layout Geometry of an Organelle

The SCMOS ⁶ nand organelles code (as inserted into the source program *organelles.l*) is listed in Table 8

The (*layout-nand-organelle*) function calls the actual layout of the SCMOS nand organelle. The name of the geometrical description of the SCMOS nand organelle is *nand2*. The SCMOS nand organelle is transformed from CIF code to L5 code by invoking the function (*cifsave*) in the program *cifdef.l*, listed in Appendix A, and concatenated into the *organelles.l* file. The NMOS (*layout-nand-organelle*) was commented out with the LISP command (*comment*).

During the execution of the input program, *nand.mac*, the *nand* organelle is instantiated and converted to an *item*. The *defsymbols* macro converts the layout geometry to an *item*. It assigns a *symbol-id*

symbol-id = (nand2 81)

⁶The SCMOS layer list is described in detail in the "text" file under */vlsi/berk83/doc/magic/scmos* directory on the VAX 11/780.

TABLE 8
CODING OF SCMOS NAND ORGANELLE

```
(defun layout-nand-organelle (drive ratio)
  (nand2))

(defsymbol nand2
  ;; The function defsymbol has the following syntax:
  ;; (defsymbol name arguments code)
  ;; The name of the defsymbol is 'nand2'.
  ;; The arguments are 'nil'
  ;; The code is the merging of a list of rectangles
  ;; and marks
  ;; that form the complete geometric description
  ;; of the organelle.
  nil
  (merge items)
  (merge
    (rect layer xmin ymin xmax ymax)
    (rect 'CWP -26 -12 -12 14)
    (rect 'CWP -24 -14 -14 -12)

    .
    (mark name x y layer attributes)
    (mark 'GND -19 -18 'CMS nil)

    .
    (mark 'IN2! -26 -3 'CPG nil)))
  ;; X and y are the
  ;; coordinates where the mark is positioned.
  ;; Attributes are used to direct other program sections to the
  ;; portion of the item they should operate on, e.g. river.
```

to the code. When a defsymbol is called, it automatically generates an *item* out of that code. It creates a *symbol* from the *item* of the form:

symbol = ((nand2 81) -28 -19 2 18&)

(symbol ID left bottom right top points internal-symbols nest-level tree)

The *symbol* form is placed on the *-L5-symbol-list*.

-L5-symbol-list = ((nand2 81) -28 -19 2 18&)

The defmacro *defsymbol* determines if the organelle is on the *-L5-symbol-list*. If it is not, it adds a new *symbol* to the *-L5-symbol-list*. It creates the *symbol* by the function (*create-symbol item symbol-id*). The *code* of a *defsymbol* is transformed into an *item*. That is, the code consisting of a list of *rectangles* and *marks* is processed into an *item* of the form:

nand organelle item = (-28 -19 2 18 ((& -19 -18 CMS nil)))

2. Organelle Extensions

The connection points on organelle's are stretched by three functions and are examined using the SCMOS NAND organelle example. The connection points that are now discussed will be in NMOS layers as this insertion is in HYBRID mode.

a. Output Connection

The (*layout-output-connection*) function extends the output terminal of the SCMOS nand gate by appending a metal wire with a poly-cut (NMOS) at the end. The bus tracks are wired in polysilicon layers. The output signal from the SCMOS nand organelle is passed to an internal bus to the *port-output* unit. The output terminal on the SCMOS nand organelle is located on the top right of the organelle. In general, if the unit's bus number is not zero ($< > 0$) i.e., (...-2 -1 1 2 ...), or the unit is a *port-output*, an output extension is built onto the organelle's output. The equality (=) and not equal ($< >$) cells are assigned bus numbers of zero (0) because they do not propagate their output signals to the datapath and thus no extension is built on the output terminal of the organelle. To find the location of the output on the organelle or port or register, the following LISP code is executed:

```
(let (output (car (call-list gen-form (cons 'output tail))))  
  gen-form = (lambda (info bit word-length drive ratio)  
    (cond  
      ((eq info 'output) '(30)) ...&&)
```

The output is placed at the head of the list called tail. Recall that the tail for an organelle is composed of the fields (bit# #bits drive ratio).

```
tail = <(output 0 4 -1 (4 4))  
info = output
```

The metal wire extension and its poly cut are marked for connection to the internal bus by the *tip* defstruct defined as

```
(defstruct tip  
  input (unit# argument# operand#)  
  output (unit#)  
  left ()  
  right ())
```

The *input* and *output* cases of the *defstruct tip* refer to bus connections to the units in the datapath. *Left* and *right* cases refer to connections made by the signal net list from the pads to the terminal connections on the left edge of the internal layout. The execution of the (*layout-output-connection*) function results in the return of an *item*. Our example, returns the following item:

```
output-connection item <- (33 25 37 36 (((0 (output 1))) 33 36 CPG nil))
```

((rect CMF 33 25 37 36) (move (symbol-call 2) 33 36)))

The *item* states that a metal wire (NM in NMOS) 4 lambda wide and 11 lambda high along with a poly-cut (NMOS contact) (identified by a call to its position on the -L5-symbol-list (symbol-call 2)), placed at the end of the wire is constructed. The poly-cut is identified for connection by a bus *tip*. The output tip is identified as (output 1) where 1 is the unit#. Figure 3.6 illustrates the topology of the output extension as defined by the above item. The poly bus line is not part of the extension.

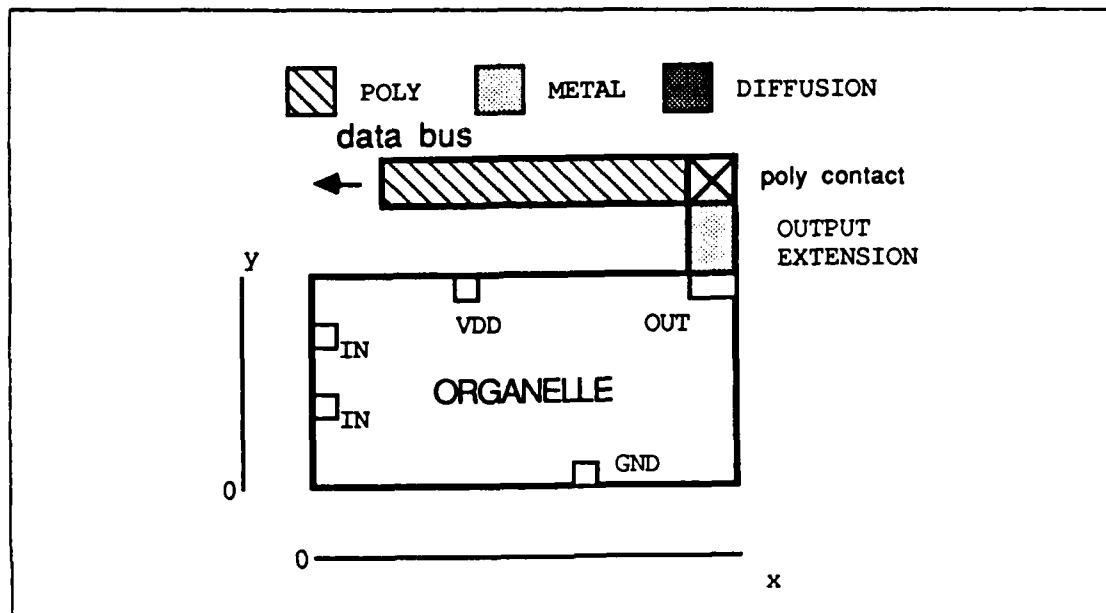


Figure 3.6 Output-Extension.

b. Lower Extension

The lower extension applies to the **MSB** of the unit. If the bit-slice of a unit has a *drive, test, or control line*, a wire is extended from the bottom edge of the MSB of the unit. A metal ground wire from the MSB organelle is connected to the ground rail of the skeleton. The *drive, test or control lines* are extended in polysilicon wire for connection to the control section via the river router. For the HYBRID nand circuit there are no *drive, test or control-lines* requiring extension, simply the ground wire is extended. The execution of the lower-extension function returns an item of the following form:

lower-extension item <- (7 -10 11 0 nil nil (rect CMS -10 11 0))

The ground wire is 4 lambda wide and 10 lambda high. That is, a 10 lambda extension is made from the organelle to the ground rail on the skeleton frame. Figure 3.7 illustrates the topological layout of the lower extension.

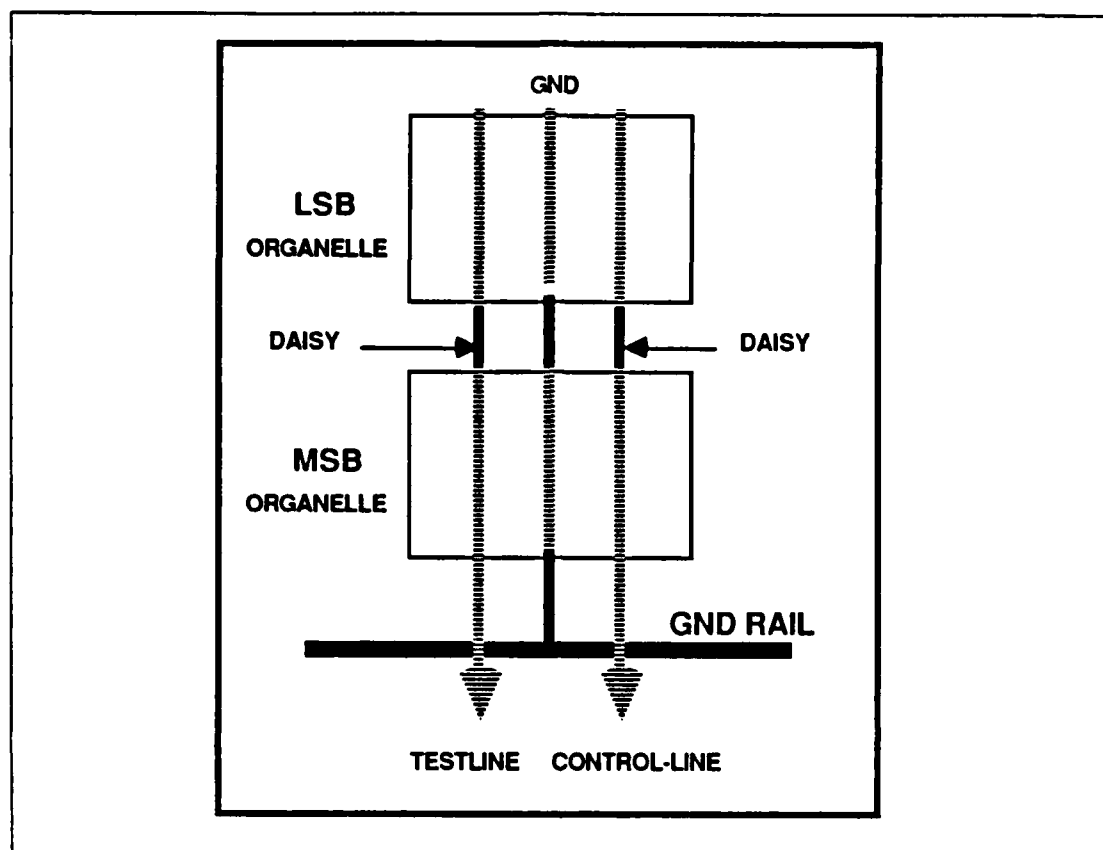


Figure 3.7 Lower Extension.

c. Upper Extension

The (*layout-upper-extension*) function extends power, and ground lines through the length of the unit and connects these wires to the skeleton. The LSB bit-slice unit has its Vdd connected to the Vdd rail of the skeleton. The remaining organelles are connected together along the vertically running Vdd and gnd wires. Any values that require daisy-chaining through the word-length of a unit have their locations in the *daisy* attribute of the *gen-form*. A *daisy* line can be a clock, carry-out, or some other output that is propagated through the unit. The locations of the Vdd, gnd and daisy terminals is found by querying the *gen-form*. The layout of these

connections is returned in the form of an *item*. For example, in the HYBRID nand circuit:

```
upper extension item <- (16 25 20 44 nil nil (rect CMF 16 25 20 44))
```

Figure 3.8 illustrates the topology of the upper-extension. A metal wire 4 lambda wide by 19 lambda high is generated from the Vdd point at the top of the LSB organelle to the skeleton power rail.

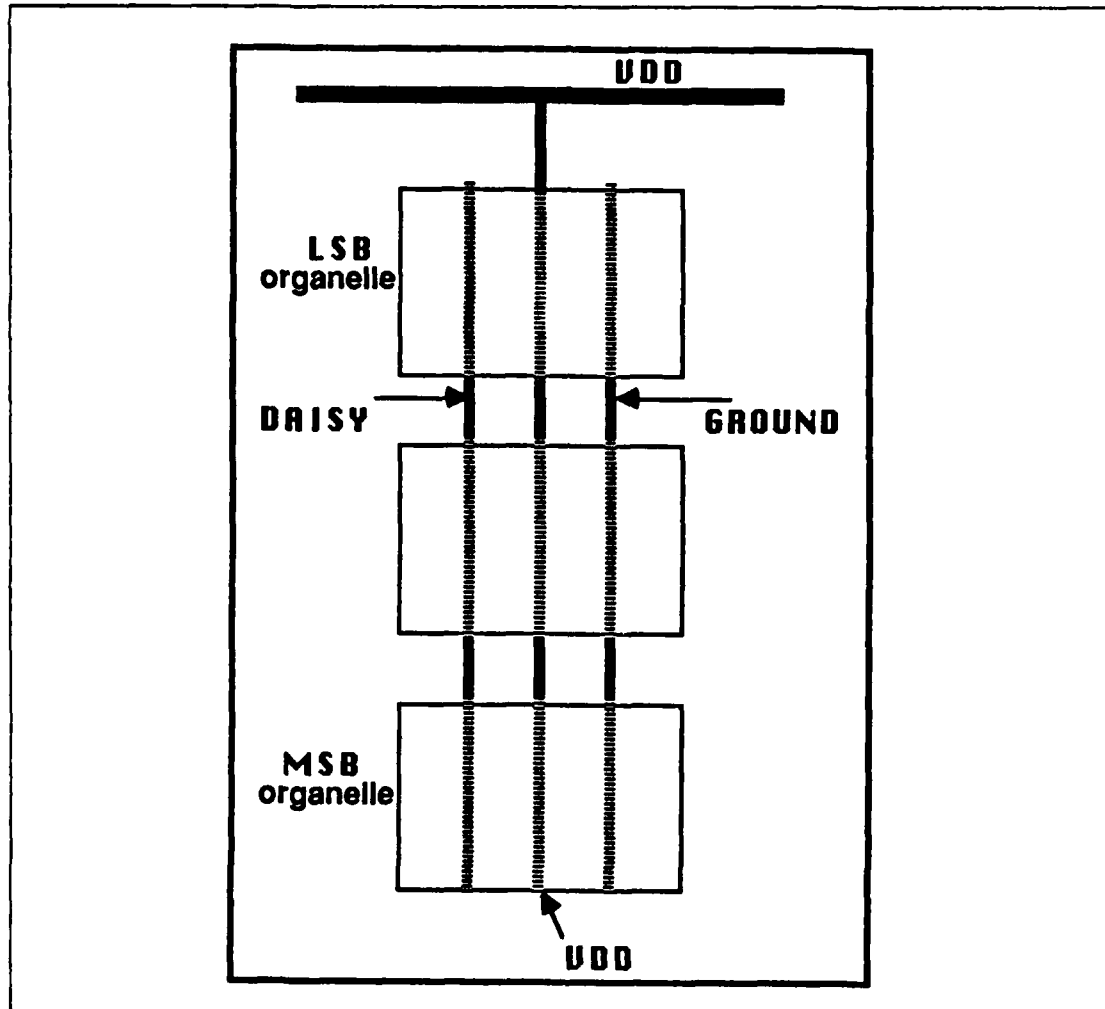


Figure 3.8 Upper Extension.

3. Layout Gen-Form

The (*layout-gen-form*) function merges the organelle along with its extension (stretch) lines and returns the resultant item called *organelle-part* to the (*layout-organelle*) function.

E. PORTS

Data is communicated between the datapath and the external world through ports. Ports are parallel buses of wires of the same width as the largest organelle in the *unit-list*. There are three types of ports: *input*, *output* and *internal*. Pins are dedicated to these I/O ports. The maximum number of pads available on a MacPitts chip is set at 64. If a port is defined with a pad number greater than 64 an error occurs. This restriction in the code can be easily removed in the (*process-<pin-name>-definition*) functions in the source program *prepass.l*

1. Port-Input

Input ports are created by the (*layout-mpx*) function. If inputs to a unit-type come from more than one source, a multiplexer is used at the input of an organelle to allow selection of which input to pass to the organelle. Simple inputs are passed from the bus along a single metal wire to the organelle. A river generated diffusion wire connects the metal wire to the inputs of the organelle. Port-input is not a separate unit but is generated in the *mpx-part* of the unit structure. The *mpx-part* for *unit#1* of the HYBRID nand circuit is

```
unit#1 mpx <- (((port-input a)(port-input b)))
```

The *mpx* specification consists of two *operands*, (*port-input a*) and (*port-input b*). Four types of *mpx-part* configurations are available (*mpx0 mpx1 mpx2 mpx3*). The (*layout-mpx*) function determines the number of *constant-bits* in the *mpx* specification. A constant specification is of the form (*constant number*). The (*get-constant-bits-from-mpx*) function recursively processes the multiplexer argument

```
argument <- ((port-input a)(port-input b))
```

The (*get-constant-bits-from-argument argument bits*) determines if a *constant* operand is in the *multiplexer* specification. If no *constant* is found in the specification a value of () is returned. The (*layout-mpx*) function determines if the *constant-bits* equal () or () t | () t | (t) | (t t). If the *constant-bits* are null as in the case of the HYBRID nand circuit specification, the (*layout-mpx0*) is invoked. A 4 lambda wide metal wire is generated along the input edge of the organelle with an extra reach of 6 lambda. An

NMOS poly-cut is placed on top of the wire for connection to the bus wires. An NMOS diff-cut is placed on the bottom of the wire. The river router will route a diffusion wire from this diff-cut to the inputs of the organelle. This is repeated for the second input wire to the organelle. Figure 3.9 shows a topological layout of the inputs to the organelle.

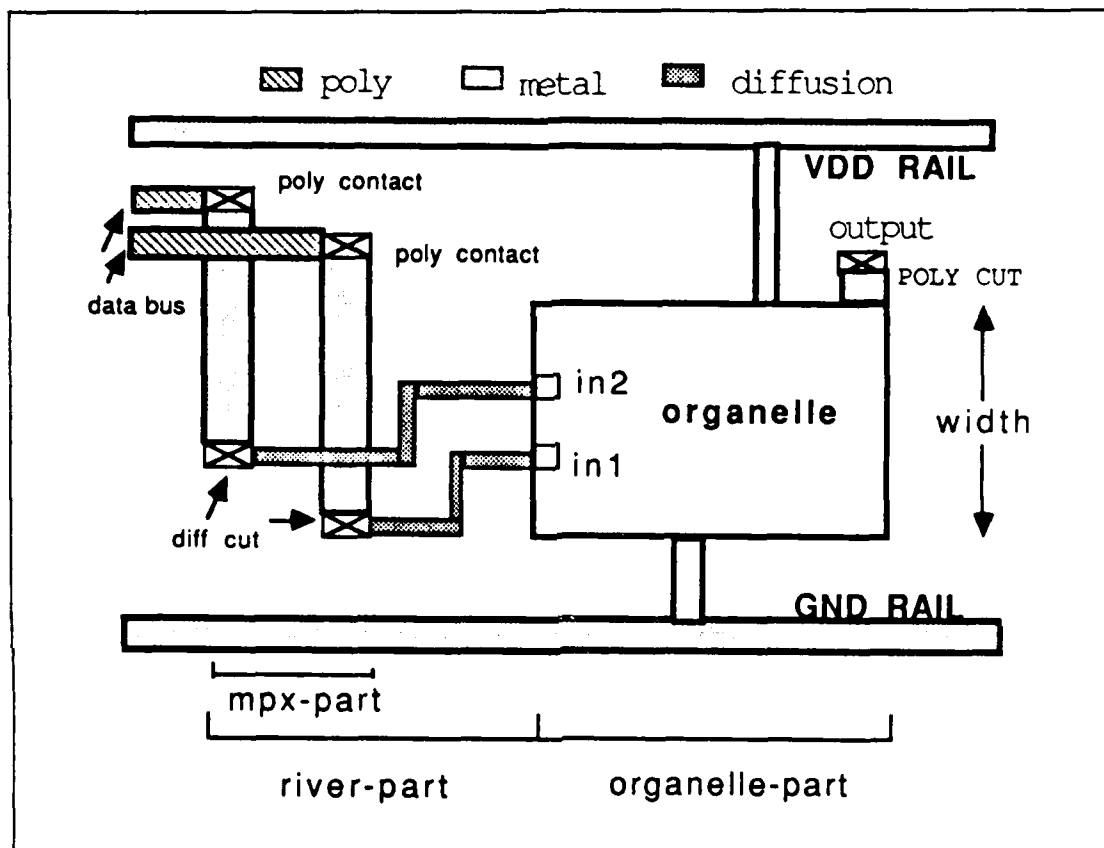


Figure 3.9 Port Input Topological Layout.

River routing between the mpx-part and organelle-part is accomplished by determining the output terminal points on the mpx and the input terminals on the organelle. The output points on the multiplexer are placed in a list called *left-bank*, and the input points on the organelle are placed in a list called *right-bank*. The (*river*) function is then invoked. Table 9 lists the code that produces the river routed wiring between the multiplexer and the organelle. Figure 3.10 shows the stipple plot of the wiring crossing the channel. A 2 lambda wide diffusion (NMOS ND) wire crosses the

channel from the input wires to the two inputs of the organelle. An *item* is returned containing the layout of the port-input.

TABLE 9
CODING FOR THE RIVER ROUTER

```
(let ((left-bank (get-output-connections-for-mpx (mpx-of
unit) #bits))
;The input locations of the organelle are found by querying
;the gen-form of the organelle.
(right-bank (call-list gen-form (cons 'inputs tail))))
;The river router is invoked by the river function.
(river-part (river 'ND 2 0 left-bank right-bank))
(river layer width stretch left right).
```

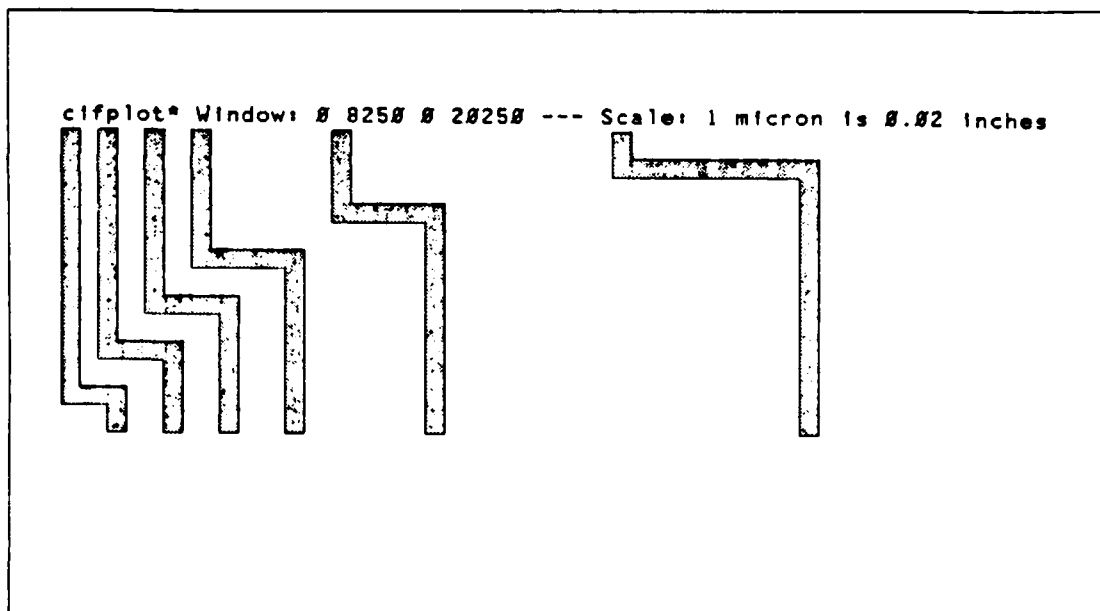


Figure 3.10 Stipple Plot of a River Connection.

2. Port Output

The second unit in the HYBRID nand circuit datapath specification is a port-output.

unit#2 <- (port-output c (internal))

A *port-output* in NMOS is simply a diff-cut with a metal wire extension.

The same extensions are applied to the *port-output* structure as to an *organelle* or *register*. The information to properly space the port-output is contained in its *gen-form* located in the source program data-path.1. The *port-output output* terminal is extended to connect with the internal bus lines. That is, the (*layout-output-connection*) function stretches a metal wire with an NMOS poly-cut on top to connect to the bus wire. There are no power, ground or control connections to the *port-output*. The *port-output mpx* specification is found by the (*mpx-of unit-list*) function

unit#2 mpx <- (internal 1)

where the *bus#* of the *mpx* is 1. The (*layout-mpx0*) invokes the (*layout-singleton-operand-list*) function. A 4 lambda wide metal wire is generated along the input edge of the *port-output* with an extra reach of 6 lambda. An NMOS poly-cut is placed at the end of the wire and marked with a *tip* for connection to the bus. An NMOS diff-cut is placed on the bottom of the wire for connection to the input of the port-output by the river router. The topological layout of the *port-output* unit is shown in Figure 3.11.

3. Port-Internal

Port-internal follows the same architecture as the port-output. It consists of a diffusion contact whose input is taken from the *mpx-part*. A specification for an internal port is of the form

**(port-internal sequencer- < name > -next-state -2
(((constant 1))((constant 0))))**

The value -2 is the organelle bus number. The *mpx-part* specification is

mpx ::= (((constant 1))((constant 0))))

The *constant-bits* (1 0) invoke the function (*layout-mpx2*) which generates a multiplexer with two input control lines and one output line that is fed to a register cell's input. The *organelle-part* of the port-internal unit is a port-output cell. A multiplexer is hierarchically constructed from sub-gate elements called *odd-operand* and *even-operand*. An even operand is a transistor with two contacts, a diff and poly cut. An odd operand is a transistor with one contact, a diff cut. Figure 3.12 shows a stipple plot of the odd and even operand NMOS structures. These structures allow the construction

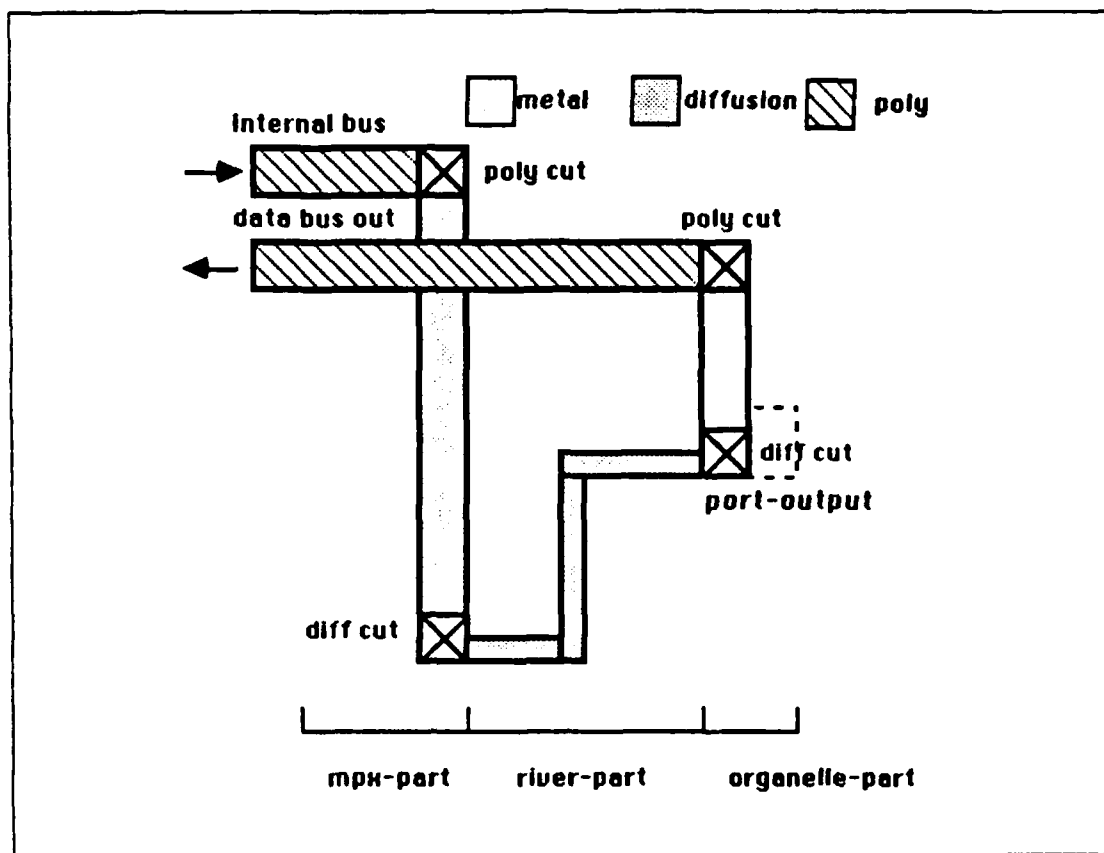


Figure 3.11 Port-Output Unit.

of 2:1, 3:1 and N:1 multiplexers. that are gated by one or more control lines. If the values in the multiplexer specification are constants, these values are hard wired into the inputs of the multiplexer. If the specification contains an (internal bus#) form, the multiplexer takes its input from the local data bus . The value of the bus# matches the organelle's bus number and indicates the source of the signal. Figure 3.12 also shows a topological layout of a 2:1 multiplexer.

F. DATA-PATH LAYOUT

The (*layout-organelle*) function merges the multiplexer (*mpx-part*) the organelle (*organelle-part*) and the channel (*river-part*) and places them side by side. The resultant *item* is returned to the (*layout-organelle-list*). This function recursively processes each subsequent bit of the unit, stacking the bits in the vertical direction. The (*layout-unit-list*) function recursively processes the next unit until all units have been transformed into one *item*. The resultant *item* is passed to the (*layout-data-path*) function. The

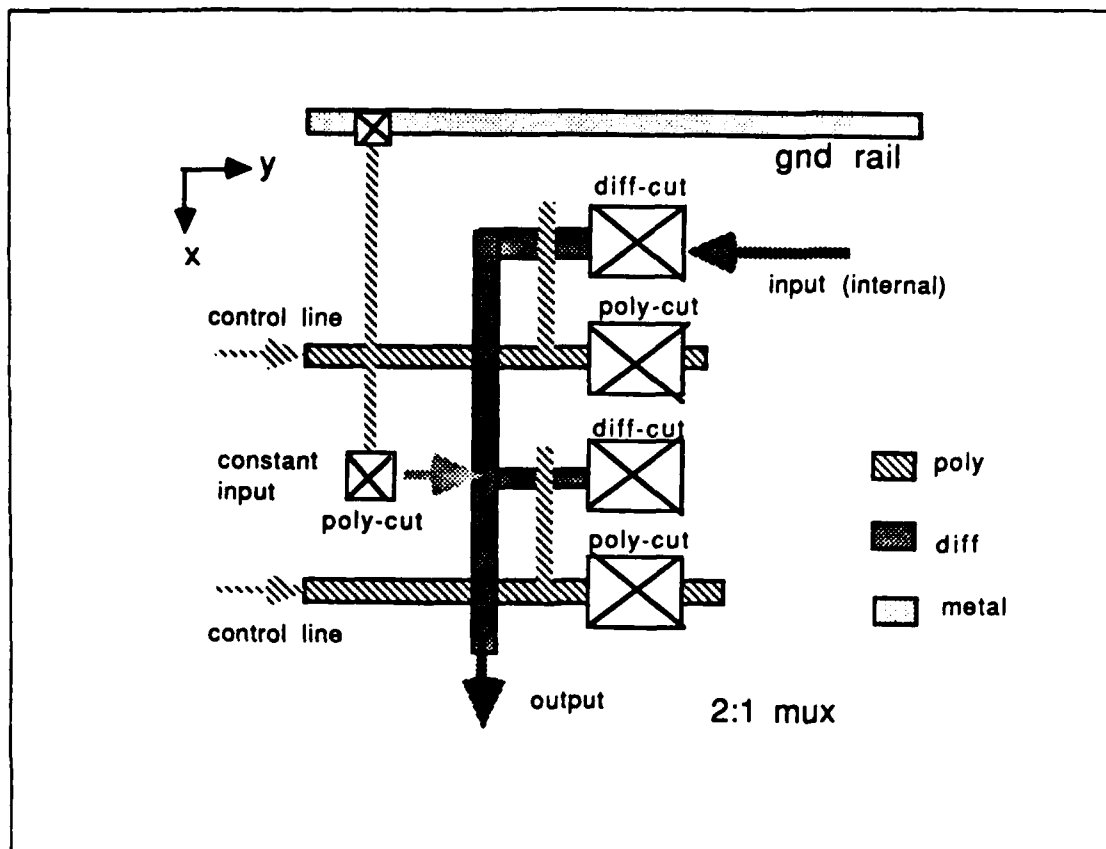


Figure 3.12 Topological Layout of an NMOS 2:1 Multiplexer.

buses are generated at this level by the (*layout-buses*) function. The *buses* are merged with the *item* returned by the (*layout-unit-list*) function. The resultant *item* is the *data-path-layout* that is passed back to the top-level function in *frame.l* (*layout-object*). The *data-path-layout* becomes a single item that is then placed relative to the control, flags, and routing channel. The *item* gives the bounding box on the *data-path*, its terminal points along the edges, and its internal layout. The points facilitate connection to the control unit via the river router along the bottom of the datapath box. On the left edge of the datapath are the terminal points for connection with the wires routed from the pads. The routing is performed by a *net extraction* in the (*layout object*) function in *frames.l*. The datapath has no connections on the top or right edge.

G. SUMMARY

The layout of a simple *nand organelle* and *port-output* were covered in detail with references to the code generating the layout. From these simple building blocks, an

understanding of the layout of complex units such as sequencers and their inter-relationships with the rest of the datapath can be understood. From this example, it can be seen that the datapath specification in the object file can be directly mapped into a layout of the datapath. The final step in the design process is to convert the L5 geometric layout of the datapath into CIF form. This is accomplished by the L5 function (*cifout item file title*). The argument *item* is defined in code as the output of the (*layout-data-path*) function.

```
(setq layout (layout-data-path data-path power required-width definitions))
```

The function (*layout-data-path ...*) returns a list in *item* form:

```
data-path item <- ( -14 0 80 54 ((( 0 (input 1 1 2))) 0 51 NP nil) .....&)
```

To create a CIF file invoke the command (*cifout item file title*):

```
(cifout layout 'data-path.l "Hybrid-nand-circuit")
```

This function will return a CIF file of the form *data-path.l.cif*. The CIF file can then be plotted using the *cifplot* command.

The final chip layout is produced by the function (*layout object*) and converted from item form to CIF form by the L5 function (*cifout*). A plot of the chip is generated by the *cifplot* command on the UNIX command line:

```
cifplot -s .005 -l bbox -P /work/malagon/CIF/patterns -b "Hybrid Nand Circuit"  
nand.cif &
```

A stipple plot of the chip generated by the *nand.mac* program listed at the beginning of the chapter and contained in the *nand.cif* file is shown in Figure 3.13.

Figure 3.14 shows a windowed stipple plot of the NMOS and SCMOS nand cells in the datapath. The SCMOS layout is slightly larger than the corresponding NMOS layout. The SCMOS organelle is a denser layout as compared to the simpler NMOS organelle. The width of the SCMOS datapath is stretched, that is, the *mpx*-part is positioned farther away from the organelle than in the NMOS circuit. The spacing is a function of the length of the organelle which is longer than its NMOS counterpart.

cifplot* Windows: 8 152888 8 148758 --- Scale: 1 micron is 0.004 inches
Hybrid Nand Circuit

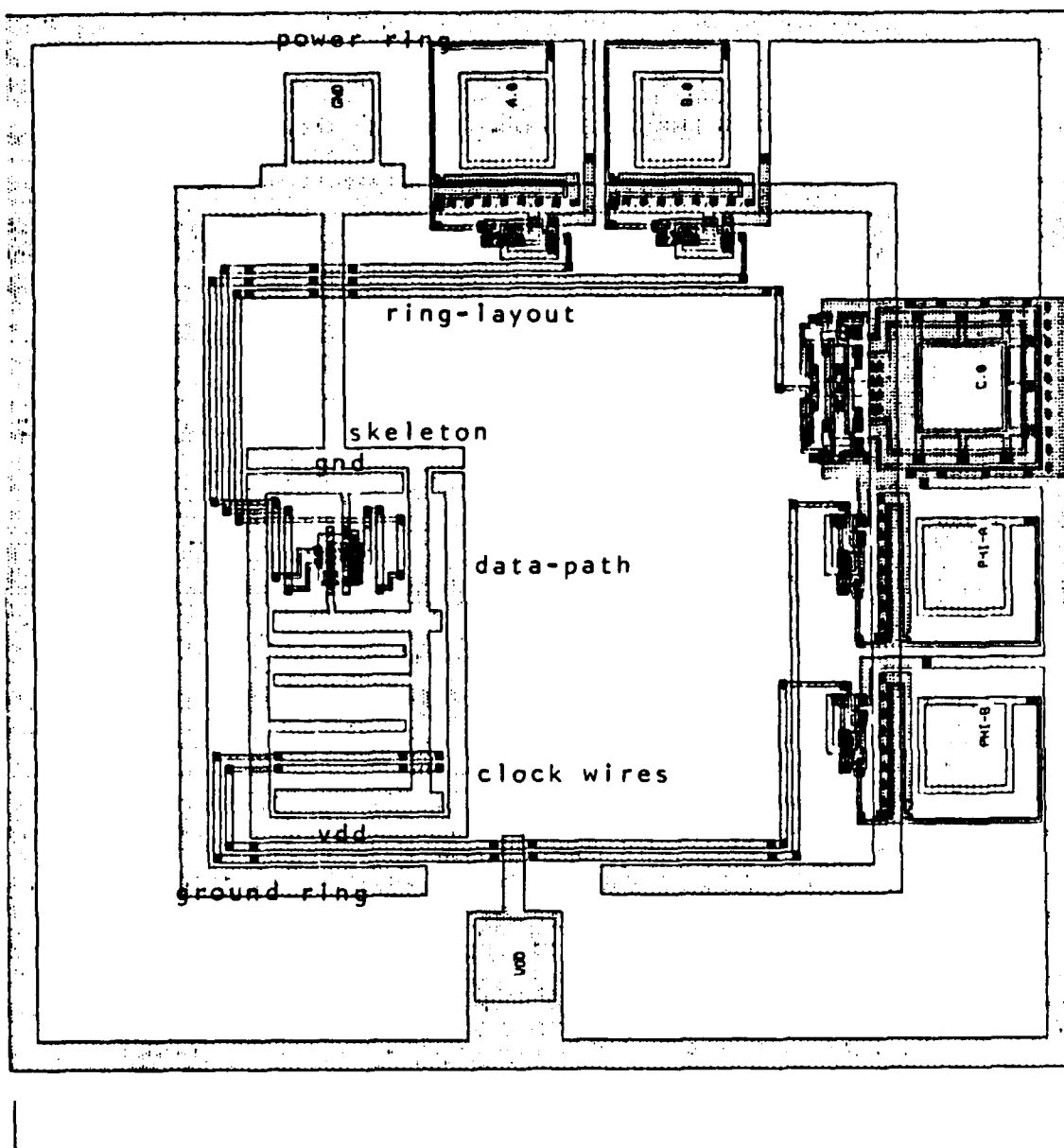


Figure 3.13 MacPitts Generated Hybrid Nand Circuit.

cifplot* Window: 35000 60000 62000 83000
scmos nand datapath

--- Scale: 1 micron is 0.015 inches (381x)

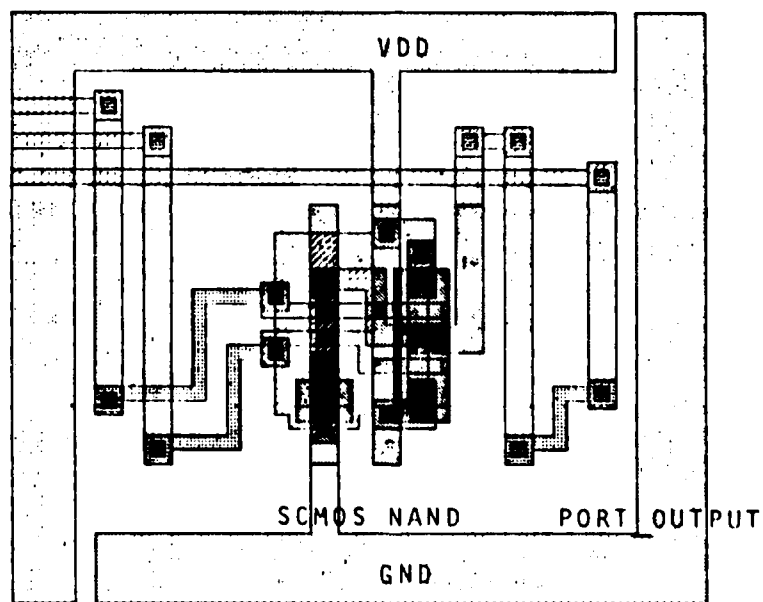


Figure 3.14 Datapath containing SCMOS Organelle.

IV. SCMOS DATA-PATH

A. SOURCE PROGRAMS CONTAINING ORGANELLES

The data-path layout generator (*layout-data-path*) references the organelle library to determine the layout (*organelles.l*) and the characteristics (*library*) of each primitive operator. The top-level arguments to the *organelle* data structure (*#control-lines*, *#parameters* *#testlines* *result?*) identify the number of inputs an organelle has, the type output and the presence of *test* or *control* signals to provide results to the control unit. For reference, recall that the organelle data structure is of the form: *organelle* *<name>* *#control-lines* *#parameters* *#testlines* *result?* (*gen-form*) (*sim-form*). Table 10 lists the top-level arguments of the basic organelles contained in the MacPitts library.

TABLE 10
ORGANELLE I/O REQUIREMENTS

organelle	#control-lines	#parameters	#testlines	result?
not	0	1	0	yes
nand	0	2	0	yes
nor	0	2	0	yes
or	0	2	0	yes
xor	0	2	0	yes
equ	0	2	0	yes
=	0	2	1	no
<>	0	1	1	no
1+	0	1	1	yes
1-	0	1	1	yes
+	0	2	1	yes
lsh rsh	1	1	0	yes
lsh-zero rsh-zero	0	1	0	yes

The *#parameters* field identifies the number of inputs to the organelle. The 'yes' in the field *results?* indicates the cells that provide outputs to the data-path. The 'no' in the field *results?* indicates the cells that provide boolean outputs to control. The fields, *#testlines* and *#control-lines* indicate the hooks to the organelle that allow passing of signals between the data-path and the control section.

The basic two-input logic units (*nand*, *nor*, *or*, *and*, *xor*, *equ* ...) provide full word output used by the data-path. Arithmetic organelles such as an adder and subtracter provide both function and test outputs. The overflow (carry-out) output of the adder organelle is a test signal that is used by control. Test units such as the equality (=) and not equal (<>) organelles provide a boolean result to the control unit to implement primitive conditions used in the *cond* construct. Their outputs are not propagated directly to the data-path. For this reason, the field *results?* has a value of 'no' assigned to it for the equality cell. Shift functions are also organelles (<< 2<< 3<< 4<< >> 2>> 3>> 4>> 8>>) whose attributes are contained in the library function (*query-handler-for-shift1-organelle*). The shift functions ((<< is left shift one bit and >> is right shift one bit) have a control-line for enable.

The goal is to compile the same design in NMOS and HYBRID technologies to verify placement and routing of the newly inserted SCMOS organelles in the framework of an NMOS chip. To accomplish this, the lower level code generation routines of the MacPitts compiler were altered. Specifically, *organelles.l* was modified to support the layout of the SCMOS organelles. The program *data-path.l* was modified to support the layout of the SCMOS registers. The *superbuffers* in *general.l* were modified to support the layout of SCMOS *superbuffers*. The contacts in *L5.l* were modified to support the layout of SCMOS contacts [Ref. 3.] The bus circuitry requires modification from the NMOS metal and poly wiring scheme to the dual metal wiring available in SCMOS technology.

An architectural change to the MacPitts floorplan was implemented. The third phase clock and its associated wiring was removed. To accomplish this change, modifications to the higher-level programs *extract.l* and *general.l* were made. *Extract.l* performs the data-path extraction and optimization. To completely remove references to three phase clocking in the datapath, the clock driver, a built-in organelle that is hard-coded for three-phase clocking, required redesign to a two-phase clock in SCMOS technology. It is important to note that although the object file may be technology independent, the layout generation routines are architecturally dependent, meaning, removal of a specified component of the chip at the top-level does not completely eliminate a layout of its supporting wiring. Removing the specification for a third clock did not remove the three-phase wiring clock wiring in the skeleton. At present, therefore, architectural assumptions are hard-coded into the layout.

B. MODIFICATION OF AN ORGANELLE

The methodology for insertion of an organelle, is illustrated in the step by step insertion of the SCMOS two input ripple adder. The SCMOS adder was designed on the Magic layout editor and stored in a file, *adder. mag*. The *adder.mag* file is converted into the universal layout language CIF by executing the following commands from the c-shell environment:

```
Magic -d NULL
:load adder
:cif write adder
:q
```

The CIF file, *adder.cif* must then be converted to an L5 file, *adder.L5*. L5 is the layout language inherent to MacPitts. The conversion process requires entry into the LISP environment which is identified with the prompt *->* symbol. The UNIX environment is identified with a *%* symbol.

The LISP environment is invoked by typing *%lisp* or *%macpitts*. However, a unique feature of LISP is the ability to dump the user's functions into the LISP environment and retain them for permanent use. The *macpitts* executable file is simply a LISP environment that includes all the functions in the MacPitts source programs. To create such an environment the user enters the lisp environment (*%lisp*), loads the source programs (*-> (include <filename>)*), executes the LISP command (*-> (dumplisp <filename>)*). The MacPitts executable file is called *macpitts* and is created by the command *-> (dumplisp macpitts)*.

To transform the CIF file to an L5 file, the conversion routine *cifdef.l* in the Appendix. is included in the *macpitts* environment and the conversion function *cifsave* is executed.

```
%macpitts
--> (include cifdef.l)
--> (cifsave 'adder)
```

Reading symbol 1

The *cifsave* function reads in the CIF file symbol by symbol, transforms the box geometrical description of the cell design into rectangular coordinates, and precedes the list with a header of the form (*defsymbol <name> nil*). Table 11 shows an example of a CIF file and the resultant L5 file after conversion.

TABLE 11
CIF TO L5 CODE

```

DS 1 1 2;
9 nand2;
L CWP;
  B 4200 7800 -5700 300;
  B 3000 600 -5700 -3900;
L CMS;
  B 1200 11100 -5700 -150;
  B 1200 11100 -3000 -150;
L CMF;
  B 2700 1200 -2250 4200;
  B 1200 1200 -7800 1500;
  B 3300 900 -4650 2250;
  B 1200 300 -5700 1650;

  B 2400 6600 -1500 -600;
94 GND! -5700 -5400 CMS;
94 Vdd! -3000 -5400 CMS;
94 OUT! 300 -300 CMF;
94 IN1! -7800 1500;
94 IN2! -7800 -900;
DF;
C 1;

```

```

(defsymbol nand2
  nil
  (merge (rect 'CWP -31 -14 -14 16)
    (rect 'CWP -28 -16 -16 -14)
    (rect 'CMS -25 -22 -20 21)
    (rect 'CMS -14 -22 -9 21)
    (rect 'CMF -14 14 -3 19)
    (rect 'CMF -33 3 -28 8)

    (rect 'CSP -10 -15 -1 10)
    (mark 'GND! -22 -21 'CMS nil)
    (mark 'Vdd! -12 -21 'CMS nil)
    (mark 'OUT! 1 -1 'CMF nil)
    (mark 'IN1! -31 6 'CPG nil)
    (mark 'IN2! -31 -3 'CPG nil)))

```

It replaces the box form of CIF as identified by the letter 'B' with the rectangular form identified by the word 'rect'. The transformation requires the parenthesized notation common to LISP. Each rectangle is parenthesized and the list of rectangles is merged into one item by the L5 function *merge*. The name of the CIF cell and the merged list of rectangular code is prefaced by the L5 function *defsymbol* and enclosed in a parenthesized list of the form (*defsymbol* <name> <arguments> <code>).

The *defsymbol* form allows the SC MOS organelle description to be manipulated within the layout-generation routines for sizing and connection information. The *defsymbol* is a macro that, when evaluated creates a symbol-number for the organelle. The symbol-number is used to call the organelle to form a hierarchical layout of a chip. The fully expanded geometry of the organelle is then reconverted into CIF code by the *defsymbol* macro and placed on the -L5-symbol-file.

An alternate symbol creating device is currently being considered wherein the CIF code is read directly for connection points and evaluated for sizing information without doing the direct conversion from CIF to L5 back to CIF. This concept is of great importance not only from the memory and time savings involved but also in the insertion of the SC MOS pads. The SC MOS pads located on the */vlsi/berk83/lib/magic/scmos* directory on the Naval Postgraduate School Vax 11/785 are quite large. The I/O pads take up to 940 lines of CIF code. To recursively process such a large file in LISP results in a *namestack overflow* condition. A new construct to remove the CIF to L5 conversion process would help tremendously in achieving a SC MOS silicon compiler of high efficiency.

1. Insertion of a SC MOS Adder Organelle

The example file, *adder.L5*, is output to the directory from which the *macpitts* environment is invoked. Two methods are available for insertion into the *MacPitts* source code of the L5 file. The first requires actual entry into the source code, recompilation of that code and then execution of the code to test the results of changes made to the source code. The second method does not hard code changes into the source code. Rather, the user enters the 'macpitts' environment and includes a file containing the modified code in that environment. LISP executes the latest version of a function. This method avoids hard coding changes and produces quick results. This method allows a preliminary look at changes prior to actually including them in the working version of *MacPitts*.

To hard-code changes into the MacPitts source code, the L5 file is appended to the source program *organelles.l* by typing on the UNIX command line *cat adder.L5 > > organelles.l*. The NMOS version of the organelle is commented out by the LISP command (*comment*). The *organelle* function contained in the library calls for the layout of an organelle when it receives the command to 'instantiate' that function. The code is

```
((eq info 'instantiate) (layout- < name > -organelle))
```

The inserted SCMOS organelle is in defsymbol form. A cell definition is created which calls the actual geometrical layout of the organelle. The cell definition is:

```
(defun layout-+ organelle (drive ratio bit)
  (cond((= 0 bit) (adderbit0))
        (t (adderbitn))))
```

where *adderbit0* identifies the bit0 adder organelle and *adderbitn* identifies bit > 0 adder organelles. The difference between the two organelles is that carry-in on bit0 is removed. The n bit adder propagates the carry-out signal to the next bit's carry-in. The source program *organelles.l* is then recompiled using the *make* facility in UNIX. On the UNIX command line, the command *make organelles.o > org.stat &* is executed. The new *organelles.o* is loaded at run time during the execution of a macpitts program

The second method of changing the MacPitts source code is to create a file that contains the new and/or changed code and include this file into the macpitts environment. For example, the file *adder.L5* and its calling function (*layout-+ organelle*) are placed in a file called, for example, *patches.adder.l* that is included into macpitts by the command *-> (include patches.adder.l)*. An input program is written to test the generation of the adder organelle as implemented in a MacPitts chip. The program is executed while in the macpitts environment in the following manner:

```
% macpitts
-> (include patches.adder.L5)
-> (macpitts adder)
```

where the filename *adder* is the name of the input program, i.e. *adder.mac*. The outputs of the execution are object and CIF files named *adder.cif* and *adder.obj*.

2. Organelle Library Changes

There are two types of organelles found throughout the MacPitts source code: functionally defined organelles and built-in organelles. Functionally defined organelles

are organelles whose names are used in the input program (.mac). These organelles are defined in the library file. The library file is loaded at run time along with the input program. Functions used in the input program (*word-not*, *word-and*, *word-nand*, *word-nor*, *= + - l+ l-*) are defined by the organelle function in the library. In essence, using the *+* symbol in an input program is akin to saying (layout-+organelle). The *+* symbol is a shortened form and the call to the instantiation of the adder function is contained in the organelle function in the library. Built-in functions are implied in the program code by the use of the *process* construct to imply registers, or declaration of ports in the definitions section of the input program (port output) (port-internal). Before an input program, <filename>.mac, is executed to test code changes, an organelle data structure is created and placed in the *library*. The built-in organelles (*register*, *driver*, *port-output*...) are not contained in the *library*. The information on sizing and connectivity are located in the layout parent source programs, *data-path.l*.

The adder function, identified in the input program as *+*, is an organelle. The *+* organelle function in the library must first be modified. The SC MOS adder organelle is larger with terminal points in locations that are different from the original NMOS organelle. The values of the length, width and terminal positions for the SC MOS organelle are listed in this function, Table 12.

The dimensions of the adder organelle are determined from a plot of the cell. To plot the cell, invoke the plot function in *plot.l* while in the macpitts environment.

-> (plot item file scale microns/lambda)

The plot function will change directories to the *tmp* directory located one level above the *vlsi* directory. The function to change directories is located in *edit.l* and should be loaded along with *plot.l* in the macpitts environment.

%macpitts

-> (include plot.l)

-> (include edit.l)

-> (plot (layout-+organelle 0 '(4 4) 0) 'adder .02 2.5)

The plot command values correspond to the following code:

(plot (layout-+organelle drive ratio bit) 'title scale 2.5microns/lambda).

The plot command will return the following information:

Changed directories to /tmp

Please wait while making CIF file

[1] 8863

-> Window -2250 4000 -8500 2750

TABLE 12
ADDER LIBRARY FUNCTION

```
(lambda (info bit word-length drive ratio)
  (cond
    ((eq info 'instantiate)
      ;;first-quadrant positions the adder organelle so its bottom left
      ;;corner is at (0 0). If the organelles inputs are not on the left
      ;;edge than use rotccw, rotccw, mirrorx, mirrorx to position.
      (first-quadrant (layout- + organelle drive ratio bit))
      ;;Dimensions and locations of terminals in lambda units.
      ((eq info 'width) 112)
      ((eq info 'length) 82)
      ;;inputs are along the left edge or the y-axis
      ((eq info 'inputs) '(90 98))
      ;;power and ground and output terminals are along the x-axis
      ((eq info 'vdd) '(17 73))
      ((eq info 'gnd) '(31))
      ((eq info 'output) '(80))
      ((eq info 'output-type) '(ratio))
      ((eq info 'conductivity) (quotient 1 0.5556))
      ((eq info '#transistors) '(19 9))
      ;;carry-out of a bit slice adder is daisy chained along the y-axis
      ;;to the carry-in of the next bit slice. The MSB has a testline
      ;;hook to control
      ((eq info 'test) '(2))
      ((eq info 'daisy) '(2))
      ;;drive is not used in the SC MOS organelles. NMOS organelles used
      ;;when their output signals were routed to I/O or tri-state pads.
      ((eq info 'drive () ) ) )
```

1 micron is 0.0666667 inches (1693x)

The plot will be 0.35 feet

The dimension and locations of the terminal points of the organelle are measured manually for each organelle. The location of a terminal point is measured to the centerline of the wire at that point. The dimensions and connection points of the organelle are determined by measuring along the x and y axis of the plot and converting the result to lambda values. Figure 4.1 illustrates the conversion process. An input program to test a n-bit adder organelle is written as in Table 13 and executed as described above in the macpitts environment.

Leaving the macpitts environment and returning to the UNIX environment a plot of the chip is made using the cifplot command. To plot dual layers, the file *patterns* located in */work/malagon/CIF* is passed to the cifplot command. The cifplot command

cifplot -P /work/malagon/CIF/patterns -l bbox -s .005 -b "adder" adder.cif

produces a stipple plot on the versatec plotter.

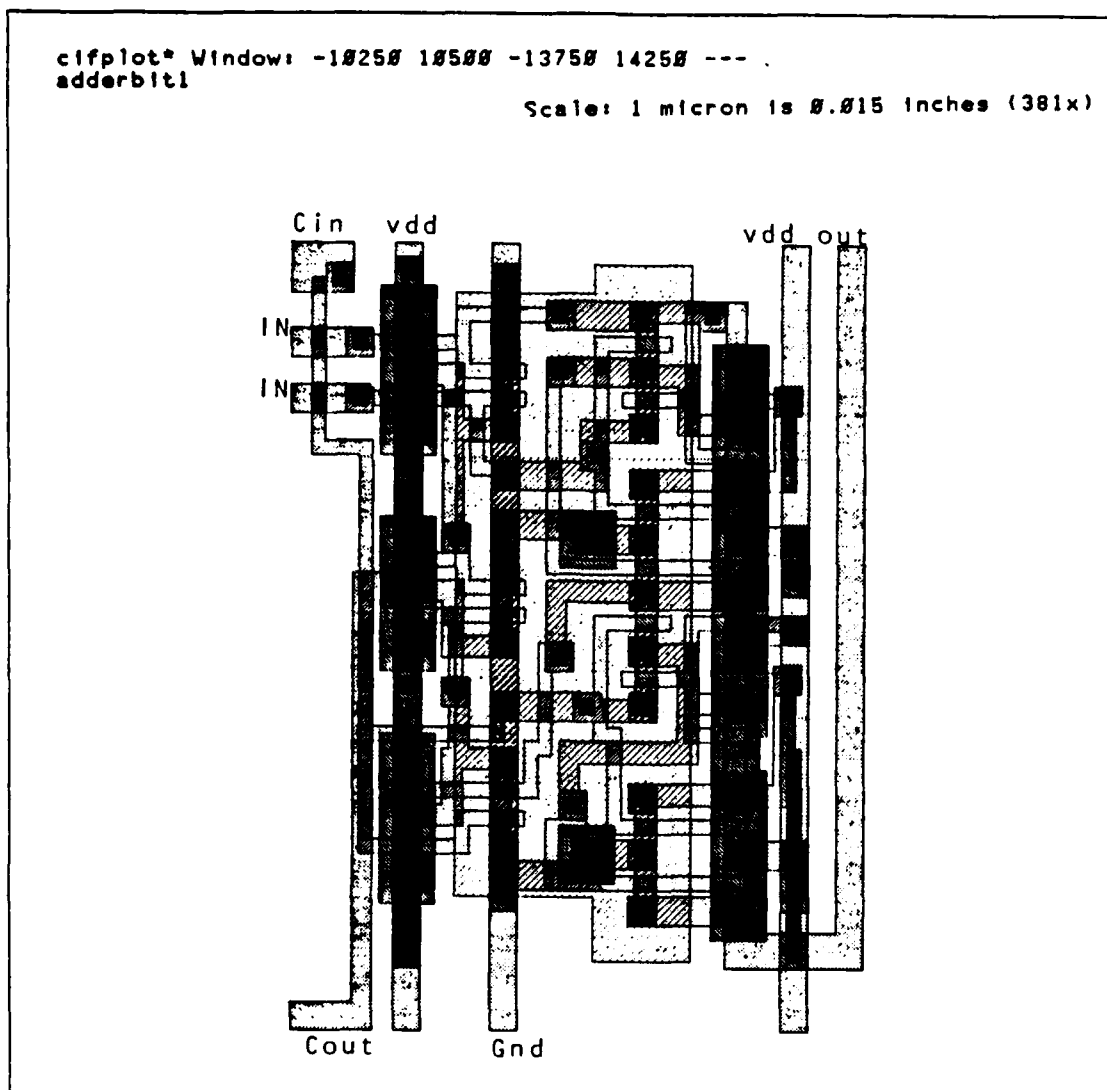


Figure 4.1 SCMOS Adderbit1 Terminal Locations.

Cifplot is invoked by typing

`cifplot {options} <filename> .cif`

The option `-P pattern-file` allows the specification of user defined layers and stipple patterns. The option `-l bbox` suppresses the bounding box around a symbol. The `-s float` option sets the scale of the plot. A MacPitts plot is normally plotted at a scale of .005 inches per micron. The `-b "text"` option prints a title on the cifplot for identification purposes. Figure 4.2 illustrates the resultant plot from the values passed to the library organelle function. Figure 4.3 shows a windowed plot of the adder cells.

TABLE 13
MACPITTS INPUT PROGRAM FOR AN ADDER CHIP

```
(program adder 4
;The word-length of this circuit is 4 bits.
;Declaration of ground/power/clock/input/output
;and signal pads.
(def 1 ground)
(def a port input (2 3 4 5))
(def b port input (6 7 8 9))
(def result port output (10 11 12 13))
(def c signal input 14)
(def carry port internal)
(def 15 phia)
(def 16 phib)
(def 17 power)
;The following code is akin to a subroutine.
;The compiler refers to this section of code
;as the component-list.

(always
(cond (c (setq carry 1))
;If a control signal is high then the internal port or bus is
;is set high i.e. to the value 1.
(f (setq carry 0)))

;Else, the internal bus called carry remains low.

(setq result (+ a (+ b carry ))))

;The value in the carry bus line is added to the value on the
;input bus line called port b. The result is added to the value
;on the bus line called port a. The sum or output from the
;second adder is placed on the port-output bus line.
```

A misalignment of the output wires of adderbit0 and adderbitn is observed from the plot. A quick translation of the adderbitn output wire on the Magic layout system is needed to fix this misalignment. To check for proper alignments, the adder organelle is windowed by using the option *-w xmin xmax ymin ymax* in the *cifplot* command line.

The length and width of the organelle form the bounding box on that organelle to which wires are connected. Measurements are taken along the x and y axis and the terminals are measure to the centerlines of their stubs. If the value of the length is greater than the actual length of the cell, the next unit will be placed further to the right of the organelle creating wasted space. If a value for width is inserted into the gen-form of an organelle that is greater than the actual width of the organelle, then the wiring does not connect to the pins of the organelle. If the length or width entered into the organelles gen-form is a smaller value than the measured value the wiring will overlap into the organelle.

Scale: 1 micron is 0.003 inches (76x)

cifplot* Win'ows: 8 168888 8 168888 ---
adder

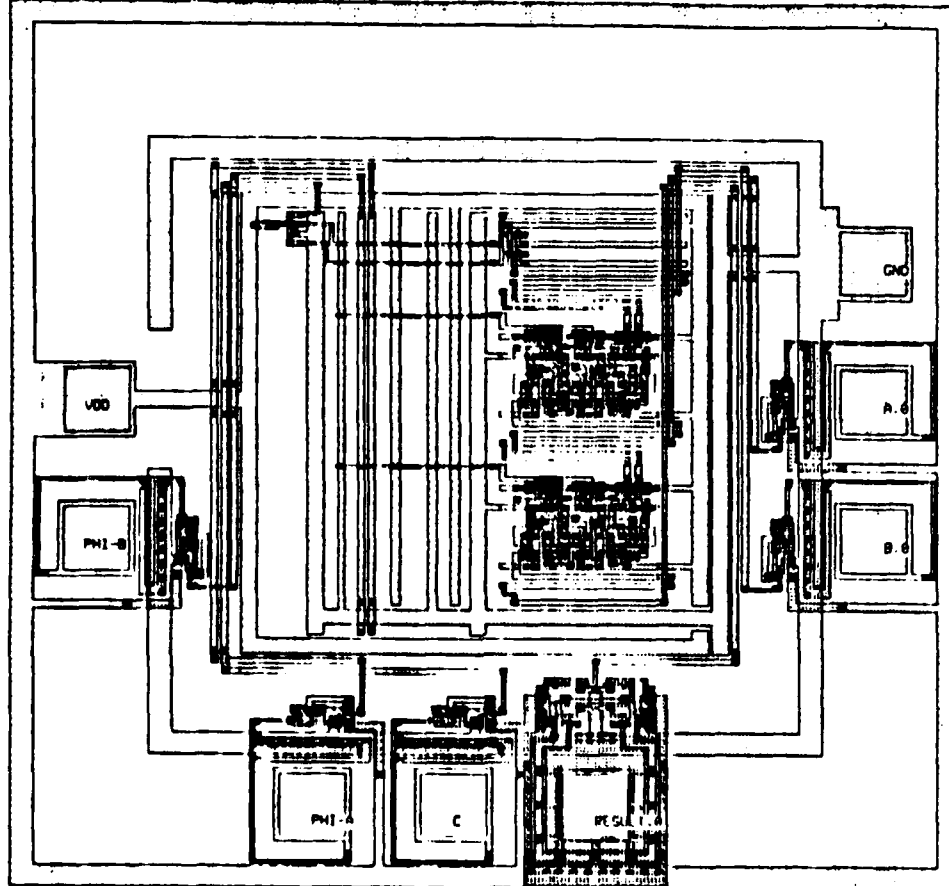


Figure 4.2 MacPitts generated SCMOS Adder Chip.

To orient the SCMOS cells for proper connections to the surrounding bus wiring, the L5 primitives: *mirrorx*, *mirrory*, *rotcw*, *rotccw* are available to flip or rotate the organelle. The bounding box of the adder organelle has its inputs taken from the left side, its carry-out from the bottom, its output from the top right side, vdd from the top and gnd from the bottom. The SCMOS designs must extend the I/O terminals to the edges of the cell for proper connection.

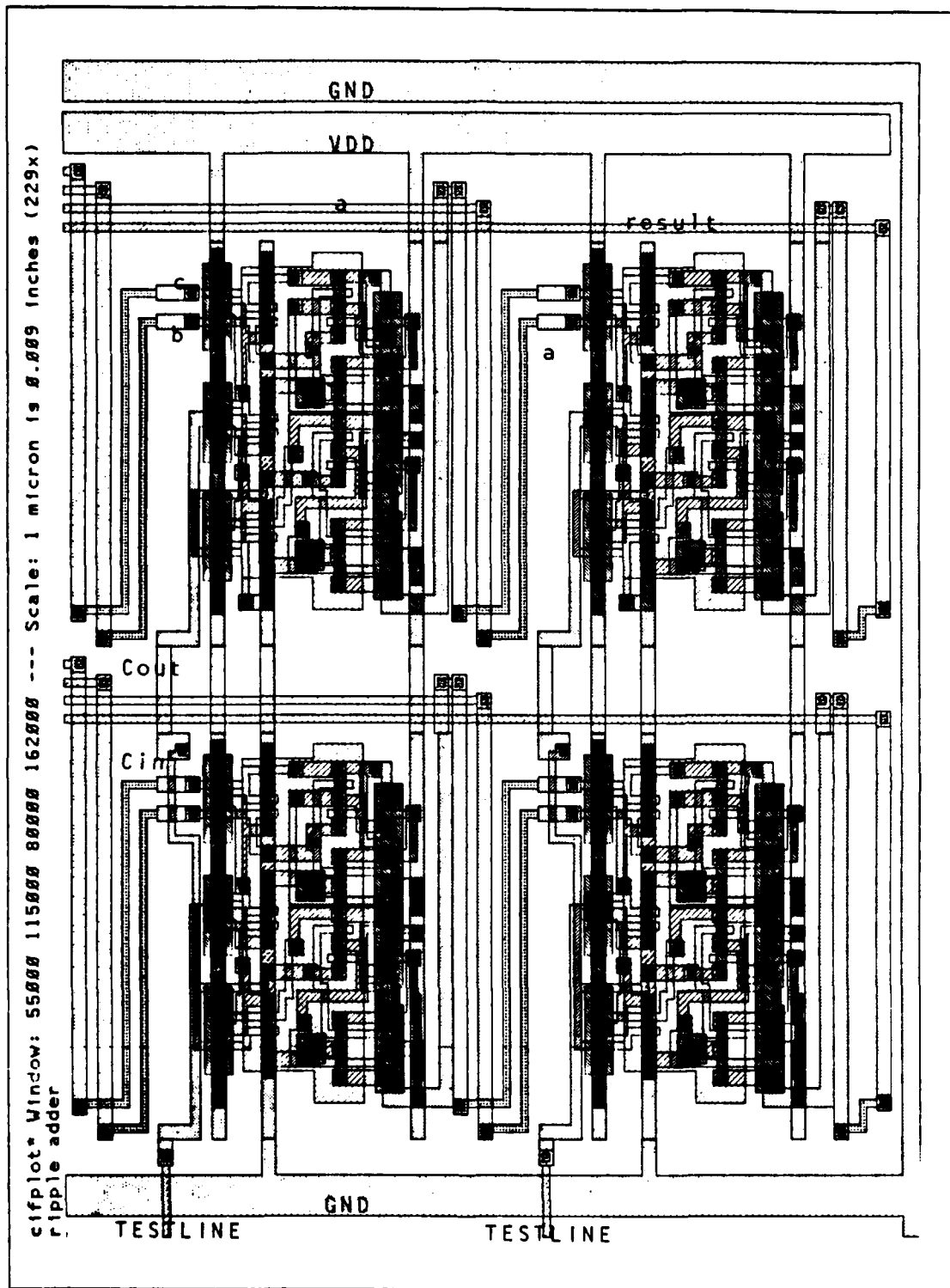


Figure 4.3 Windowed Plot of 2-bit wide SCMOS Adder Datapath.

C. SCMOS CELL LIBRARY ATTRIBUTES

A powerful attribute of SCMOS is provided by a second metal layer. This gives a greater degree of freedom in distributing global and local power ground, data and control buses in a system. First metal is the primary routing layer and will be used whenever possible.

The Mullarky SCMOS cell library designs utilize second metal for internal cell wiring and first metal for I/O terminals on the organelle. External clock, power and ground connections use second metal layers. These layers run the full length of the organelle permitting the daisy chaining of power and ground wires between one-bit slices in a unit. This information is important in determining the wiring layers for the data buses, stretch connections from the organelles, and the power and ground buses (skeleton) in the SCMOS technology. The internal power and ground buses, i.e., the skeleton will also be run in metal1. Since the Vdd, gnd lines are run in metal2 their connection to the the skeleton necessitates m2contacts at the LSB and MSB organelle or register. The internal bus wiring, that is the wires tapping off of the horizontal data buses may be in metal2 and connected m2contacts.

The structure of the SCMOS library cells does not vary as a function of drive, bit position or word-length. Drive is built into each SCMOS organelle and separate organelles exist for varying bit position. In comparison, the NMOS organelles are parameterized. This allows an organelle's structure to vary with bit position and word-length. The symbolic layout of the organelles produced less than optimum sized organelles as compared to the highly optimized SCMOS organelles.

The SCMOS technology has two types of transistors, pfet and nfet. The total number of transistors used in the chip is determined from the gen-form of the organelles. This field will have to be changed to reflect the increase in total number of transistors used in a SCMOS chip design.

D. ORGANELLES.I

1. Two Input Ripple Adder with Carry

The adder cell has three inputs, and two outputs: carry-out, and a sum output. Carry is rippled through the adder and the overflow is passed to control via a test-line. In order to form an n-bit adder, n of these elements must be cascaded with carry-out connecting to carry-in of the next bit-slice adder.

The input program that specifies an adder circuit was discussed in section B of this chapter and is listed in Table 13. This program was executed to test the adder

function. Two adders are specified in the program. The first adder adds the carry-in signal to one of the input values. The sum is passed to a second adder and added to the second input value. [Ref. 4] The carry signal is passed from the control unit to the input of the first adder via the unit called *port-internal*. The output of the first adder is passed to the second adder and added to the second data input value. The output of the second adder is the sum and is passed out of the data-path via a *port-output* unit. The .mac program generates an object file that describes the data-path as having four units: a *port-internal* to pass the carry-in signal to the first adder from control, the two adder organelles, and a *port-output*. The object specification (.obj) of the adder data-path is

```
data-path <- ((port-internal carry -1 (((constant 1)) ((constant 0))))
              (organelle + -2 ((port-input b) (internal 1)))
              (organelle + -3 (((port-input a) (internal 2))))
              (port-output result (((internal 3)))))
```

The control specification supplies two signal-input wires to the multiplexer providing a value of either 0 or 1 to the input of the adder.

```
control <- (((mpx 1 2) (nor ((primitive (signal-input c)))))
            ((mpx 1 1) (primitive (signal-input c))))
```

The topology of the resultant datapath is shown in Figure 4.4.

Negative values are assigned to the bus numbers of the individual units. Positive values are assigned to the internal bus numbers. The internal bus numbers indicate the unit from which it receives its signal. The (internal bus#) specification is located in the unit that receives the signal from the corresponding organelle bus#. That is, the specification (internal 1) indicates that the first adder organelle receives its input from the *port-internal* unit whose bus# is the value -1. The adder circuit datapath has four tracks: two input wires, one output wire and a fourth segmented wire yielding the three internal wires connecting the units together. The SC MOS adder requires 30% less area than the NMOS adder organelle in the MacPitts library. The computation time to process this chip remained the same for both organelles. A stipple plot of the SC MOS and NMOS adder chips plotted at a scale of .002 inches per micron is shown in Figure 4.5.

2. Two input Equality (EQU and =) Cells

MacPitts contains four types of equality cells: *word-equ*, *equ*, *=* and *eq..* *Word-equ* is a word-equality organelle that operates on integer inputs of the form

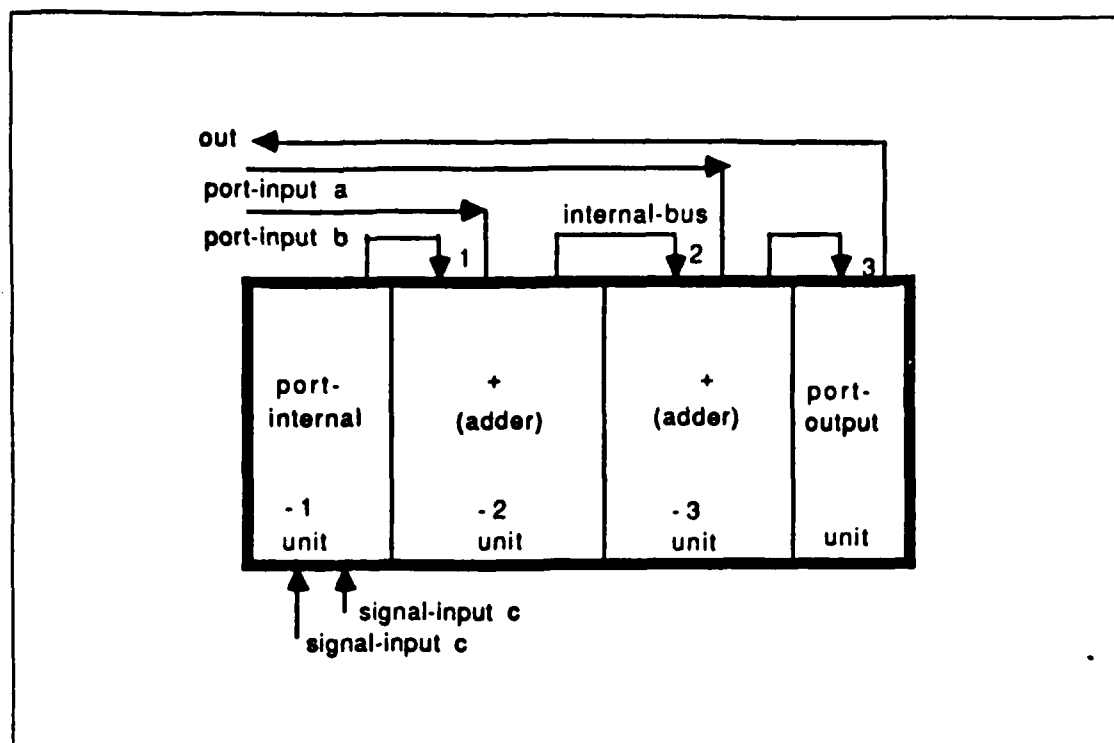


Figure 4.4 Topology of Adder Chip.

(word-equ integer integer) and returns the same result as (word-not (word-xor)) or the xnor of two integer values to the datapath. *Equ* performs the same logical function on boolean inputs of the form (equ boolean boolean ...) and returns a boolean value of 1 if the arguments are the same. The word-equality-tester organelle (=) of the form (= integer integer) determines the equality between two integer inputs and returns a boolean value of t or f to the control unit. The comparison test function *eq* determines equality between an integer and some constant in terms of bit positions. For example, (eq in "a" (5 4 3 2 1 0)), determines the equality between the value of 'in' and the ascii encoding of the character 'a' in bit positions 5 4 3 2 1 0.

The SCMOS xnor cell available in the Mullarky Cell Library matches the 'equ' function. To be used as an = function the output of the xnor cell requires that an 'and' cell be connected to its outputs to form a boolean result. Reference 5 discusses the use of a wired-or based NMOS equality (=) organelle that reduced the size of the equality cell by 51% over the original MacPitts NMOS equality organelle that used an *and* cell at its outputs. The single bit boolean form of the equality cell, the = cell and the 'eq' organelle require designs in SCMOS technology for insertion into MacPitts.

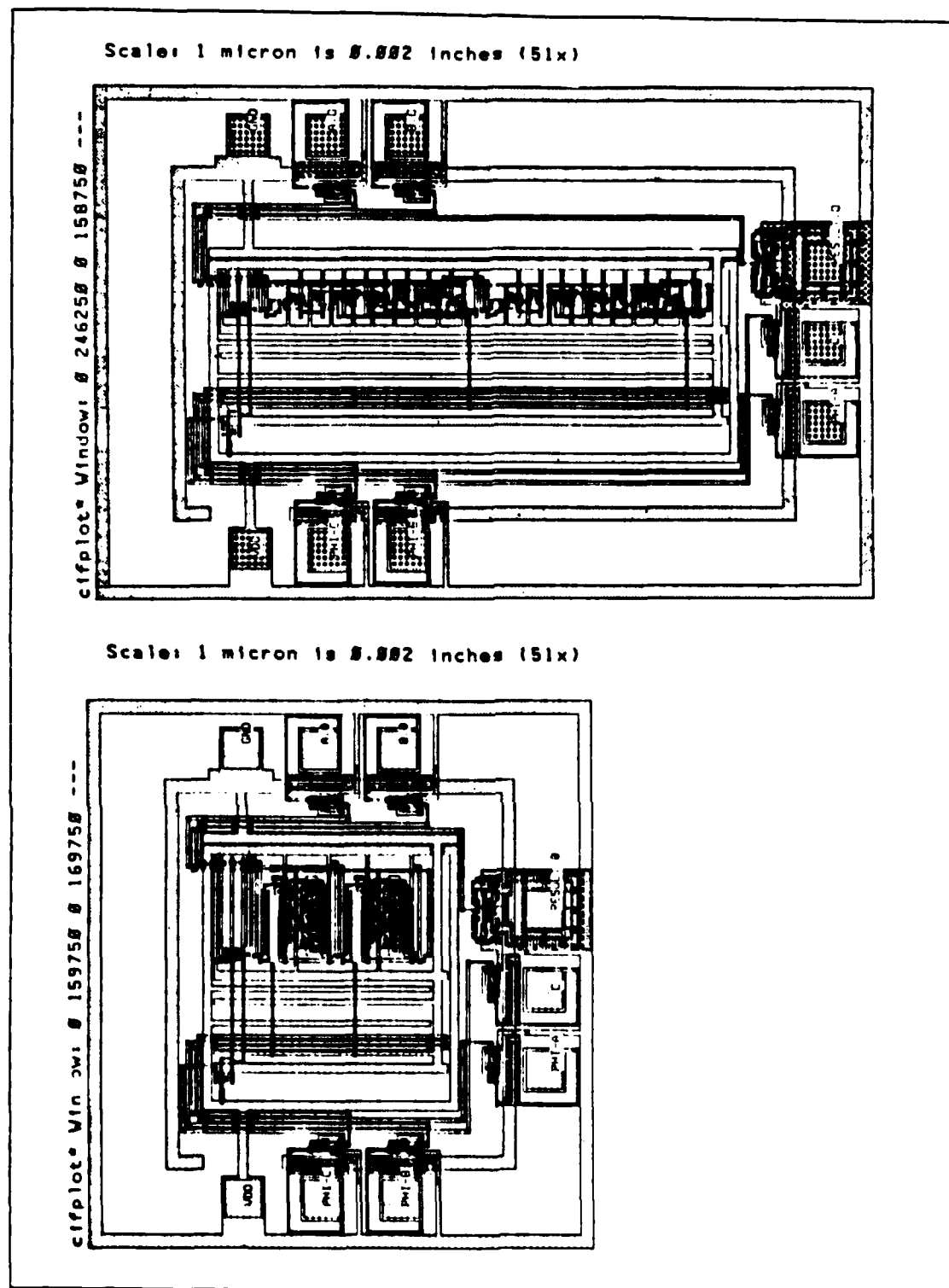


Figure 4.5 Comparison of NMOS and HYBRID Adder Circuits.

The equality (=) cell requires a bit0 and bitn cell. Since the = cell provides a boolean value of true or false and passes this output to the control unit via a testline, and does not provide the data-path with a result, an output extension is not required. The 'test' info field of the equality's gen-form will need to be updated to the location of the 'output' of the = equality cell.

An input program to produce an equality (=) cell is:

```
(program equality 4
  (def 1 ground)
  (def 2 phia)
  (def 3 phib)
  (def in port input (4 5 6 7)
  (def out signal output 8)
  (def reset signal input 9)
  (def 10 power)
  (always (cond ((= in 5)(setq out t))))))
```

The layout resulting from this program is shown in the stipple plot of Figure 4.6. The behavioral specification tests the value of the input signal for equality with the constant 5. The input to the equality cell is a port passing the in signal and a hard-wired input of the constant 5. The output is a signal passed to control via a testline. The data-path specification produced by the compiler for the program is:

```
datapath <- ((organelle = 0 (((port-input in)(constant5))))))
```

The control specification makes the testline that receives the boolean value of the equality (=) cell.

```
control <- (((signal-output out)(primitive (test-line 1 1))))
```

E. DATA-PATH.L

1. Static D-Flip-Flop Memory Element (Register)

Memory elements are used in MacPitts by control for next-state information and for storing data. Registers used to hold state information are termed sequencers. Three types of sequencers can be constructed based on the subroutine calls (go, call and return) in the input program. The basic sequencer is simply a register that receives next-state information from the control through a port-internal unit and returns state information to the controller via a bit wire. A sequencer with a counter utilizes an increment (1+) organelle and allows the controller to generate next-state information only when the machine deviates from sequential program flow. The counter and stack sequencer uses another register to temporarily store states. Figure 4.7 shows the simple 'no-counter-no-stack' sequencer. Figure 4.8 shows the 'counter-no-stack' sequencer and the 'counter-stack' sequencer in topological form.

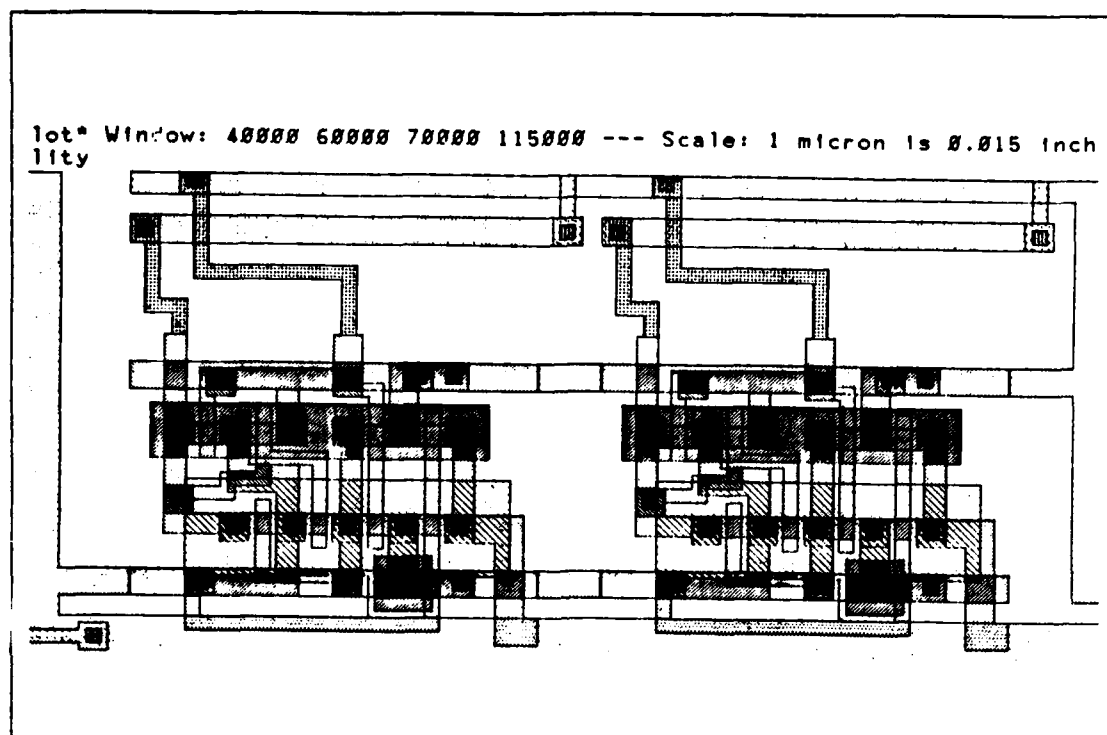


Figure 4.6 Mask Layout of a SC MOS Equality (EQU) Gate.

The register is a static memory cell, thus the storage element is not volatile and does not need to be periodically refreshed. Next-state information is passed to the register via a *port-internal* unit which is simply a multiplexer. To read the state of the cell, a control line input to the register's own multiplexer is energized which is coincident with PHI-B of the clock and the bit is read onto the control bus via a *bit* unit composed of a wire. Figure 4.9 illustrates the feedback in a register path implementing a synchronous finite state machine.

a. Bit Unit

The (*layout-bit*) function generates a metal wire along the word-length of the register. The output read from the register is passed to control via this metal wire. *Bit* returns a boolean value to control. The bit layer is easily changed to first metal in SC MOS. A bit is described by its gen-form in terms of length, width and input. The argument *bits-needed* is passed to the (*layout-bit*) function.

A bit-list of (0) produces a single wire that runs through the word-length of the register unit. The bit0 register value is passed to control. A bit-list of (1 0) produces a bit unit consisting of two wires onto which the contents of the bit1 and bit0 registers

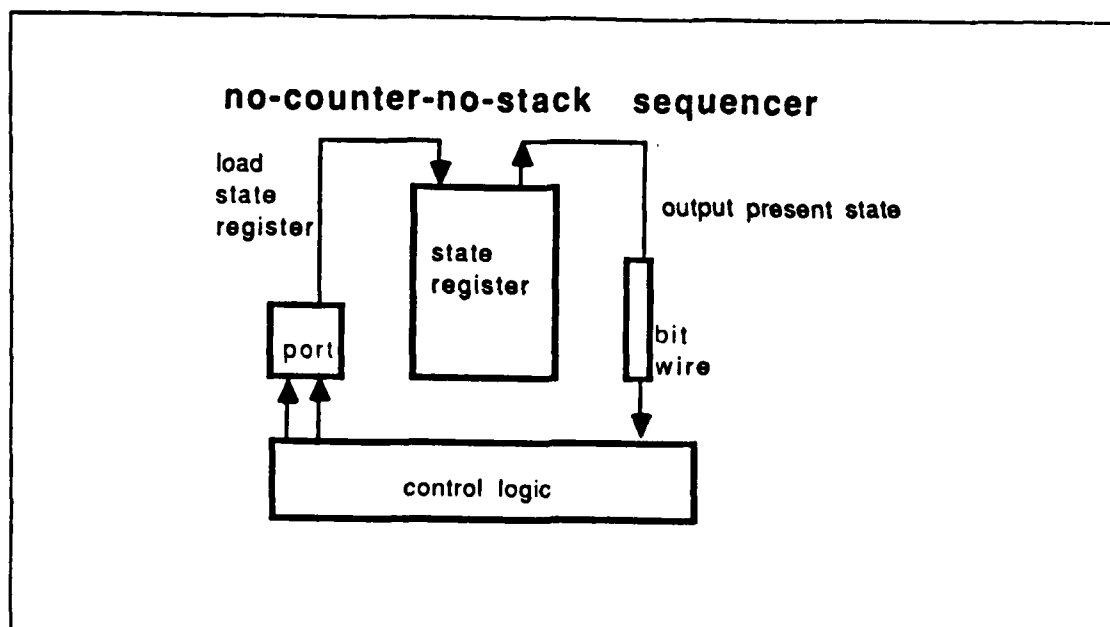


Figure 4.7 Sequencer Topology.

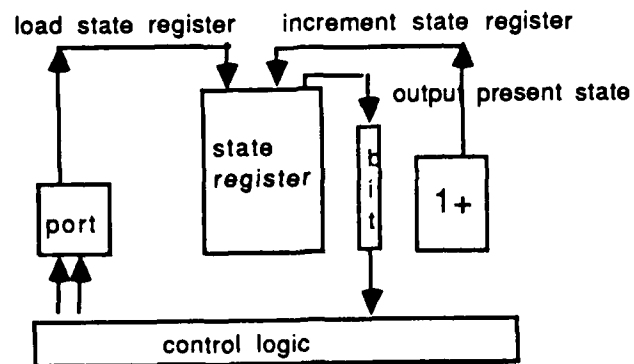
are passed to passed to control. Figure 4.10 illustrates the bit unit for a bit-list of (1 0). The number of bits in a bit-list determines the number of wires instantiated.

The instantiation of the bit unit involves the generation of a metal wire with a diff cut at its input. A mpx-part wire is generated that taps off of the bus carrying the output signal of of the register and passes it to the bit wire via a diffusion wire.

b. Register Unit

The mask layouts of the SCMOS two phase D flip-flop register cell and the NMOS register cell are shown in Figure 4.11. In comparison to the NMOS register the SCMOS register is considerably larger. Three phase clocking allows a compact layout of the NMOS register but requires an extra pin in the package and extra wiring to accommodate the third phase. The two-phase clock implemented in the SCMOS register resulted in a larger design. Removal of the extra pin and associated wiring had little effect on overall area reduction in the current fixed frame topology. The significance of removing the third phase clock pad will be seen when pads can be placed on all four sides of the chip. The impact will be pronounced when the SCMOS pads are inserted into MacPitts. Figure 4.12 shows the MacPitts layout of a 4-bit SCMOS register. The cell's output connection does not extend to the edge of its

counter-no-stack sequencer



counter and stack sequencer

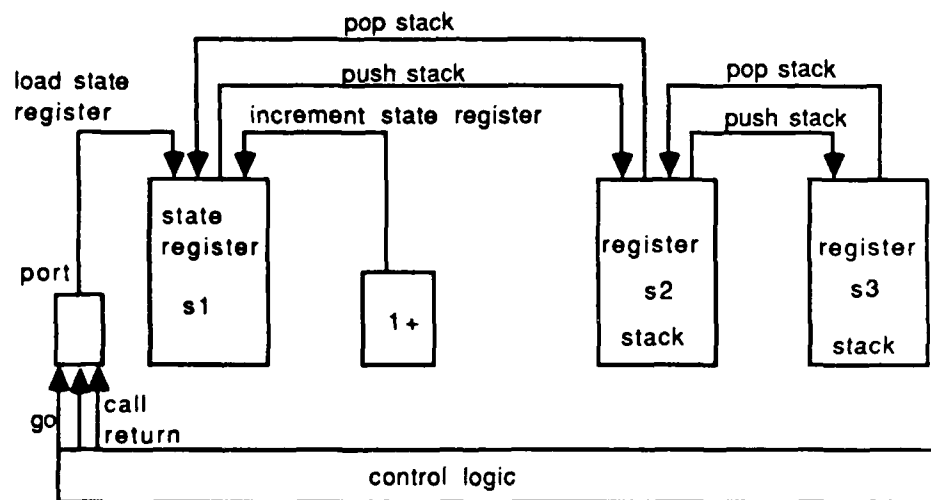


Figure 4.8 Sequencer with Counter and Stack.

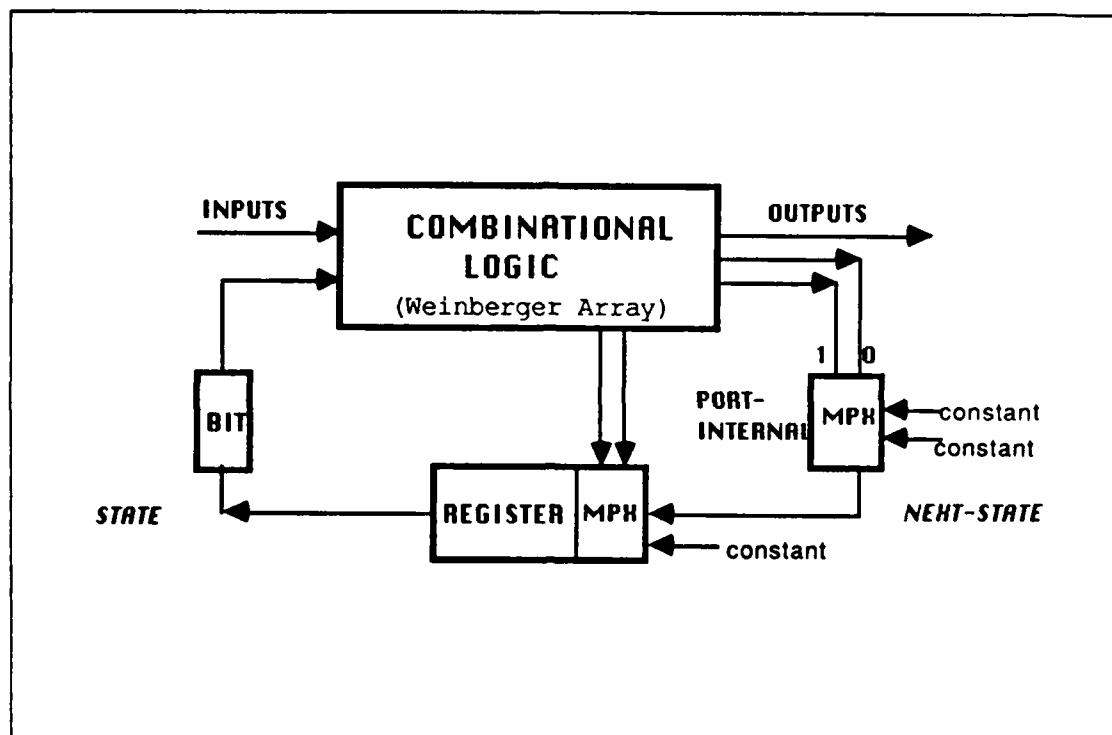


Figure 4.9 MacPitts Hardware Implementing a Finite State Machine.

bounding box causing the output extension not to butt up to the register. A quick extension on the original circuit in Magic would alleviate the problem. Figure 4.13 illustrates a windowed view of the register, the clock driver and the associated two-phase clock wiring on the skeleton.

The insertion of the SCMOS register results in a wider data-path than for an NMOS MacPitts circuit. The SCMOS register stretches the data-path in the vertical direction and shrinks it in the horizontal direction. The width of the datapath is determined by the number of bits in the data word. The length of the datapath and flags block grow as the number of datapath operations increase. Since the datapath and flags blocks grow in the length-wise direction, there is considerable savings using this SCMOS register in the length-wise reduction of the data-path for complex VLSI circuits. It is noted that the SCMOS register is the largest cell in the datapath. The datapath scales all extensions and wiring to the dimensions of the largest element in the datapath.

To test the insertion of the register, a behavioral specification was written utilizing the *process* construct. The program in Table 14 lists the behavioral specification of an equality chip with a finite state machine to store state information.

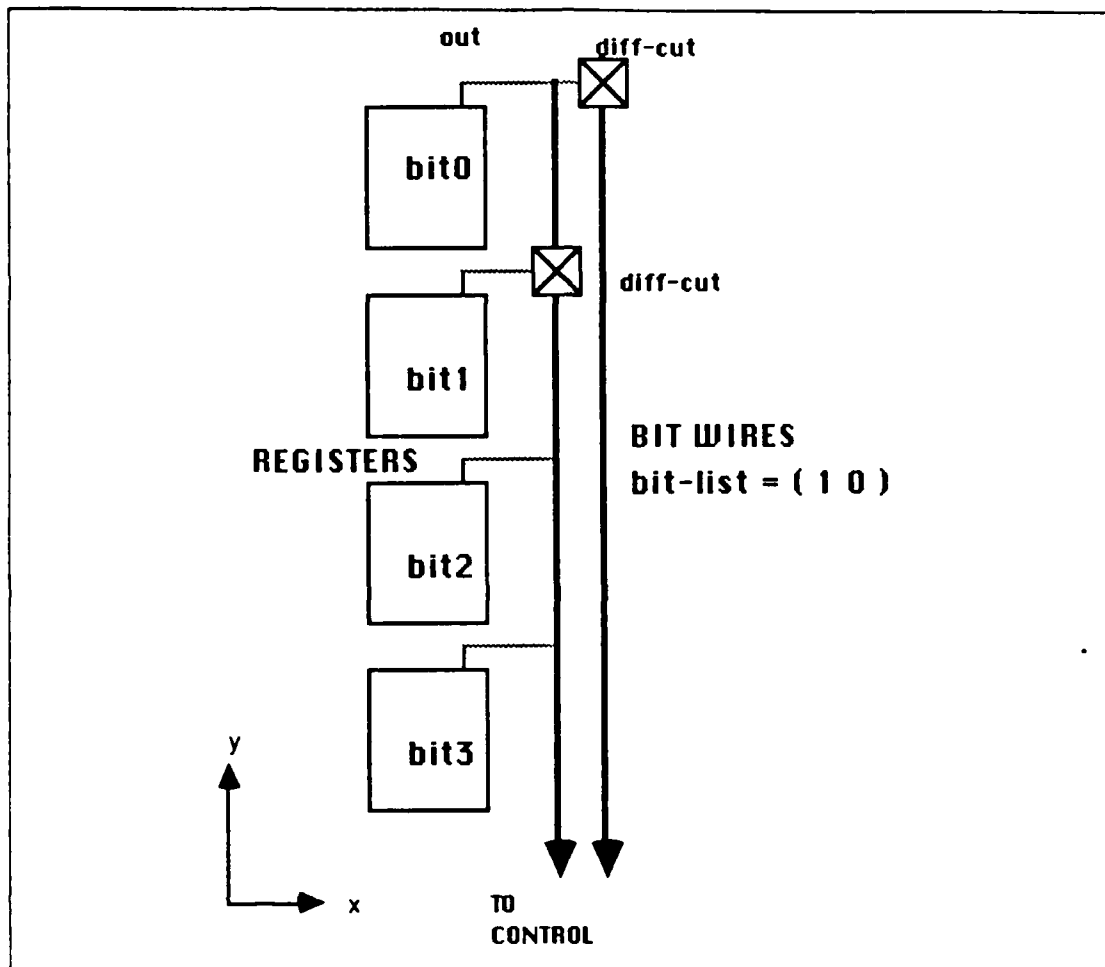


Figure 4.10 Bit-Unit Topology for a Bit-List of (1 0).

The resultant object specification is

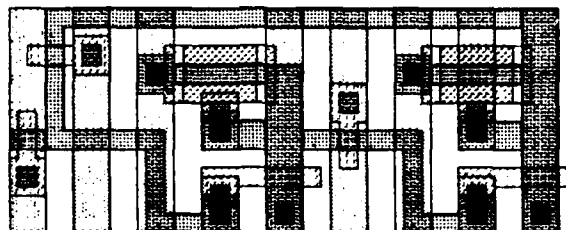
```
data-path <- ((organelle = 0 (((port-input in)(constant 1))))
  (register sequencer-equality-state -1 (((constant 0)) ((internal 2))))
  (port-internal sequencer-equality-next-state -2 (((constant 0) )))
  (bit (0) (((internal 1)))))
```

Figure 4.14 shows the topological layout of the data-path from the specification given in the object file.

c. Two Phase Clock Drivers

The SCMOS version of MacPitts will use a two-phase clocked scheme. To accomplish this, the third phase pad PHI-C and its associated wiring was removed.

cifplot* Window: 0 15500 0 6000
 nmos-register-driver=0
 Scale: 1 micron is 0.02 inches (500x)



cifplot* Window: -11600 9600 -5800 5200 ---
 scmos-register
 Scale: 1 micron is 0.02 inches (500x)

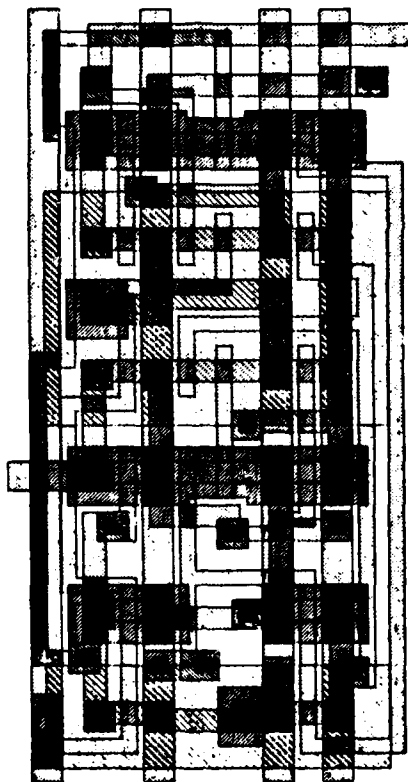


Figure 4.11 Comparison of an NMOS and SCMOS MacPitts Register.

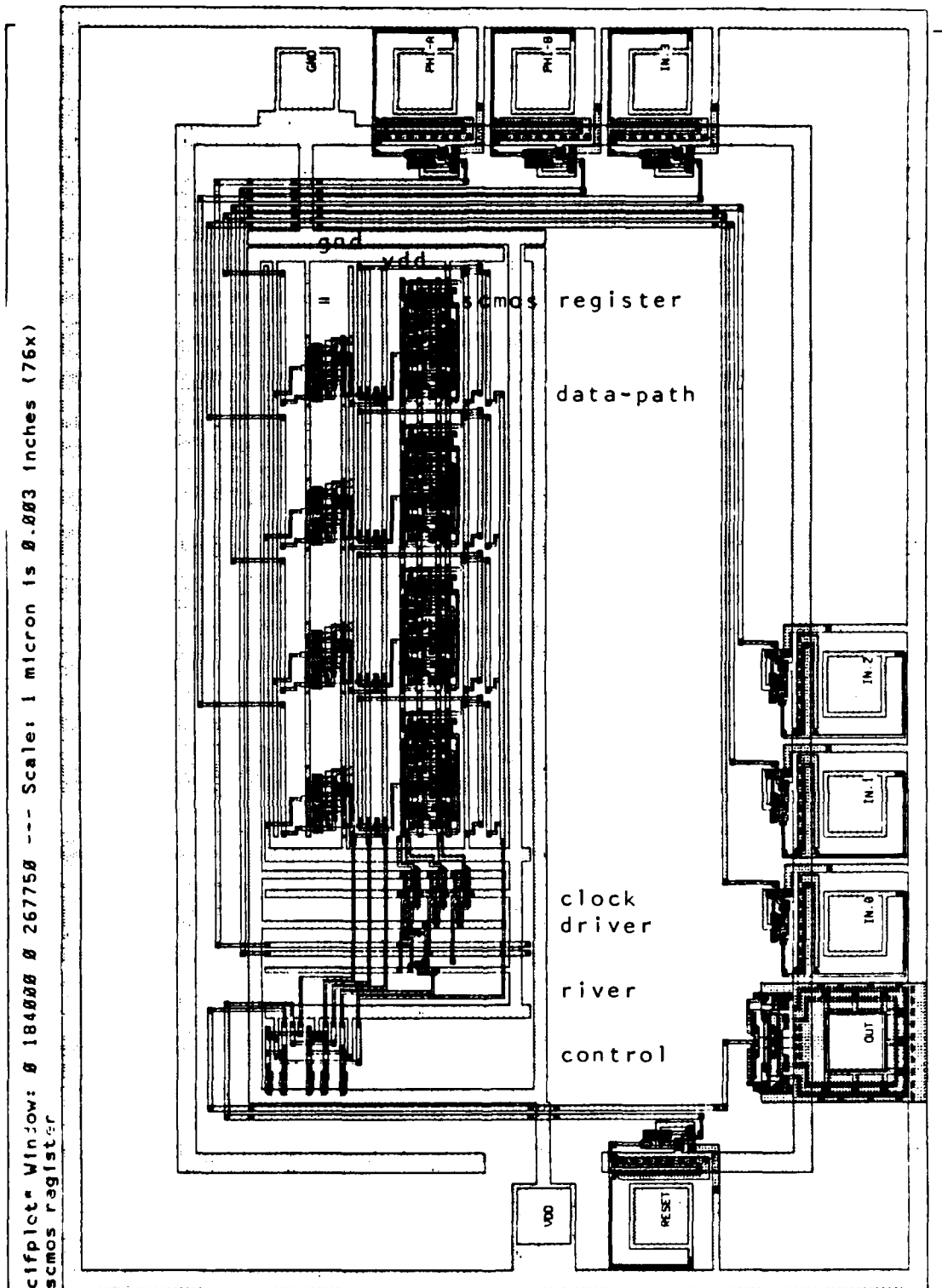


Figure 4.12 Hybrid Circuit with SCMOS Register.

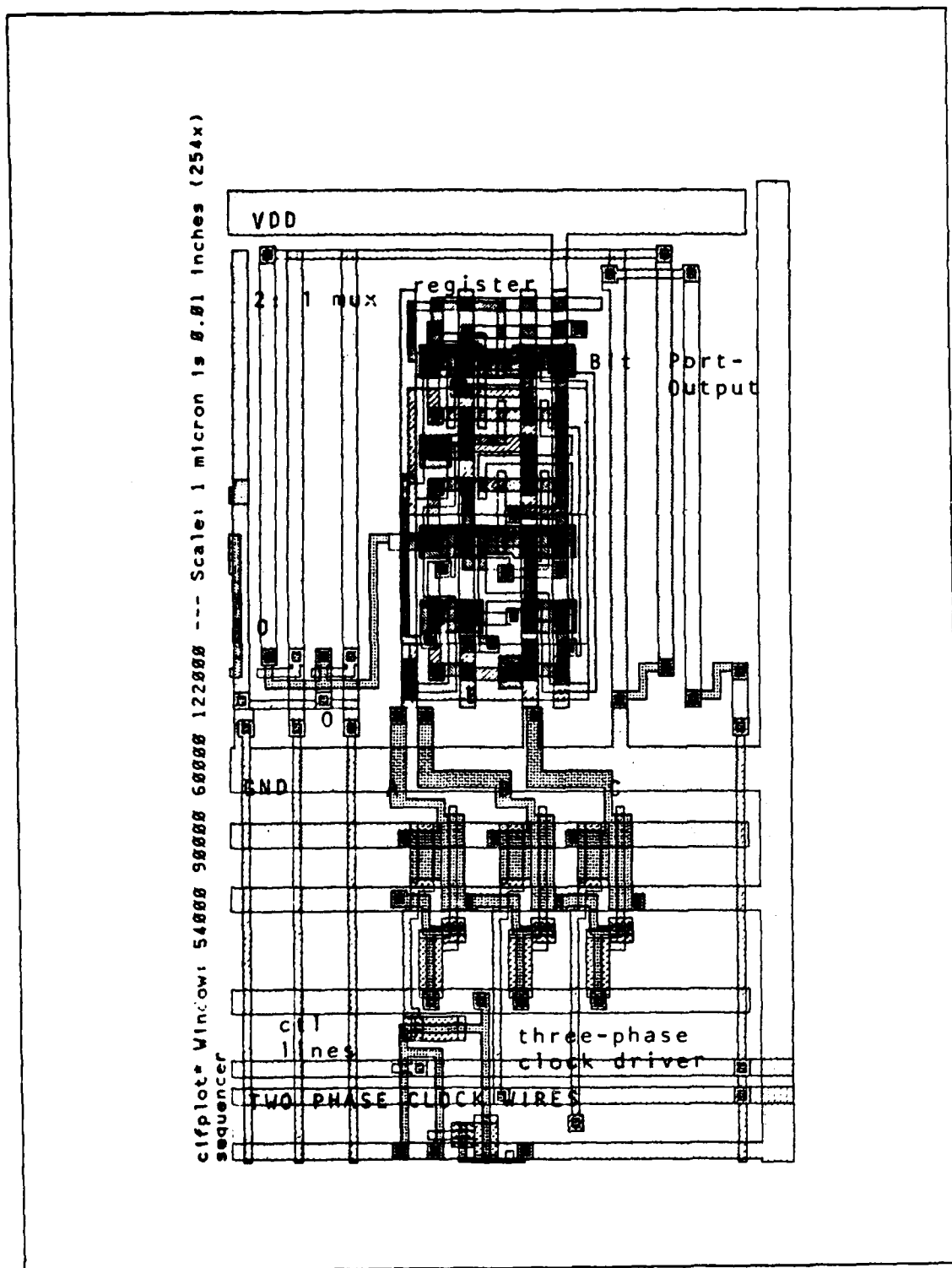


Figure 4.13 Windowed SCMOS Register, NMOS Clock Driver and Mux.

TABLE 14
MACPITTS INPUT PROGRAM FOR A REGISTER

```
(program reg 4
  (def 1 ground)
  (def 2 phia)
  (def 3 phib)
  (def in port input (4 5 6 7))
  (def out signal output 8)
  (def reset signal input 9)
  (def 10 power)
  (process equality 0
    ;(process <name> <stack-depth>
    ;<label>
    first
      (cond ((= in 5) (setq out t) (go first))
            (t (go first)) )))
```

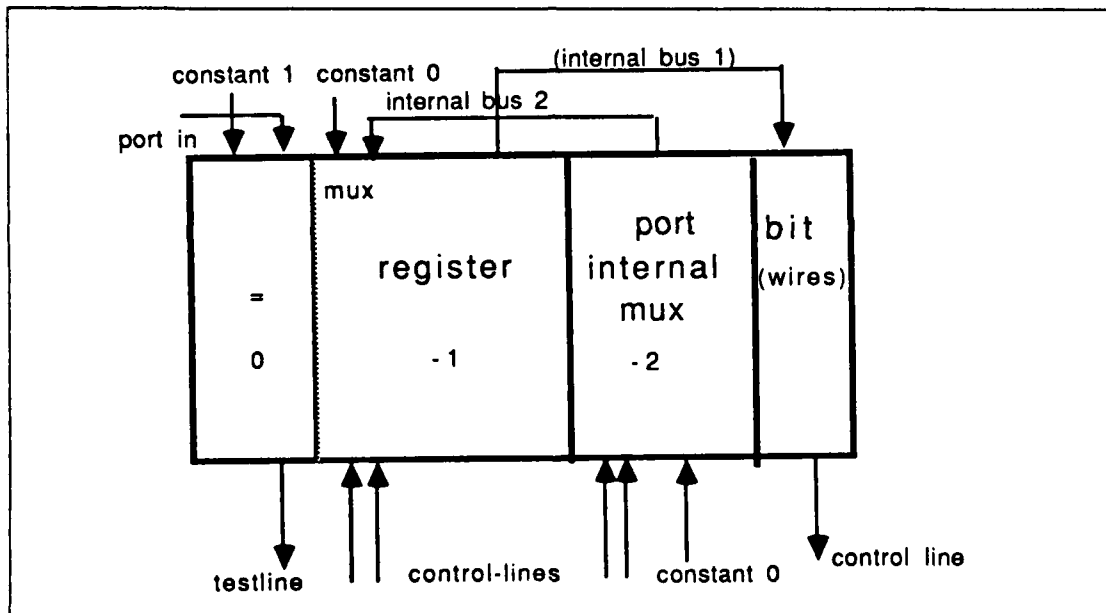


Figure 4.14 Topology of an Equality Circuit with State Register.

Within the data-path, the register and the clock drivers required conversion to a two phase SCMOS design. The SCMOS register has been inserted into MacPitts. The need exists to design a SCMOS two phase clock driver. The outputs of the driver must be pitch-matched to the clock lines on the register for correct abutment. Figure 4.15

shows the NMOS three-phase clock driver. The driver is composed of input logic for gating by the control unit, driver superbuffer and output lines. The NMOS superbuffer is also shown. The (*layout-driver*) function in the source program `data-path.l` instantiates the clock driver.

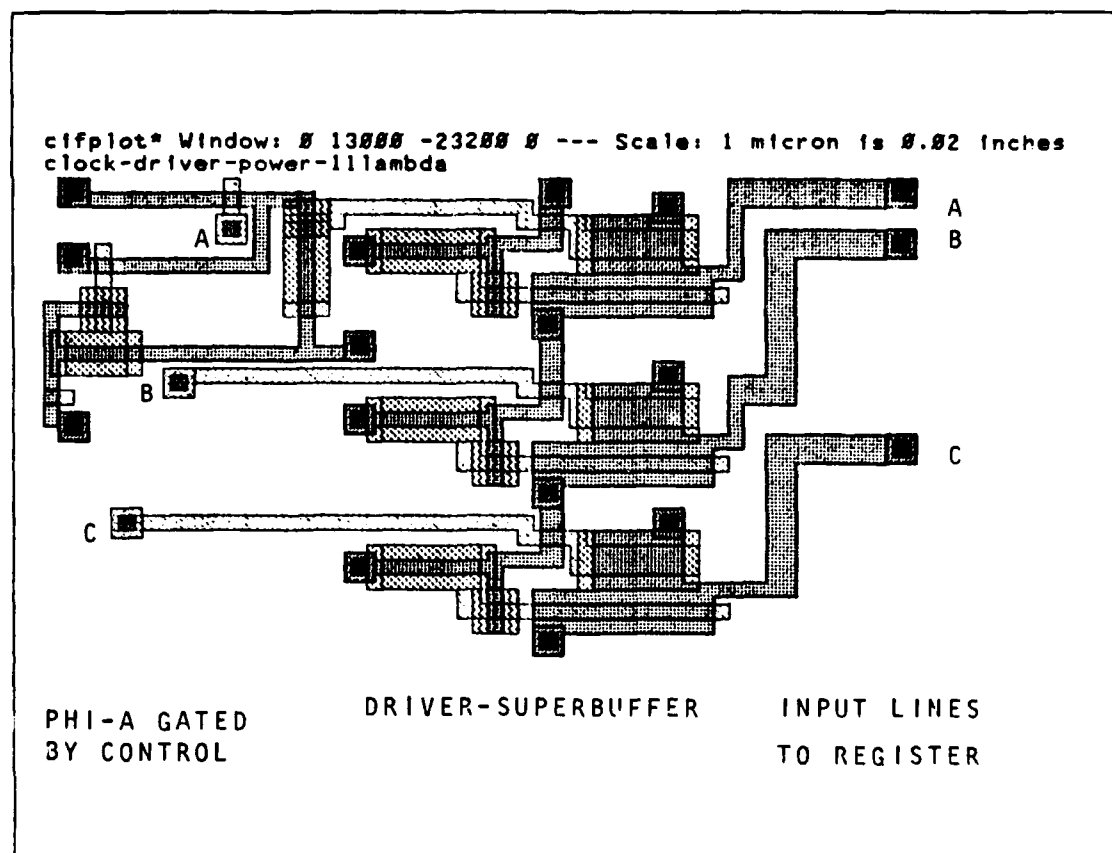


Figure 4.15 Mask Layout of NMOS Clock Driver.

2. Multiplexer

Multiplexers are fundamental to the data-path architecture. In a design specification where the predicates are simple Boolean values (true/false) these signals would be distributed via control commanding the multiplexer to connect the organelles and registers to the correct buses.

The modification of MacPitts to a microprogrammed controller requires a multiplexer specification that represents the new control type. A two input SCMOS multiplexer is available in the Mullarky Cell library (Figure 4.16), but was not inserted

into MacPitts. MacPitts hierarchically constructs a 2:1 or 3:1 or n:1 multiplexer from sub-gate units depending on the design specification. The Mullarky 2:1 SCMOS multiplexer was not designed to fit into the MacPitts topology. The NMOS multiplexer takes its input and select lines on the bottom and top of the cell. The SCMOS multiplexer should be re-designed to take its inputs from top or bottom sides to fit into the existing routing scheme.

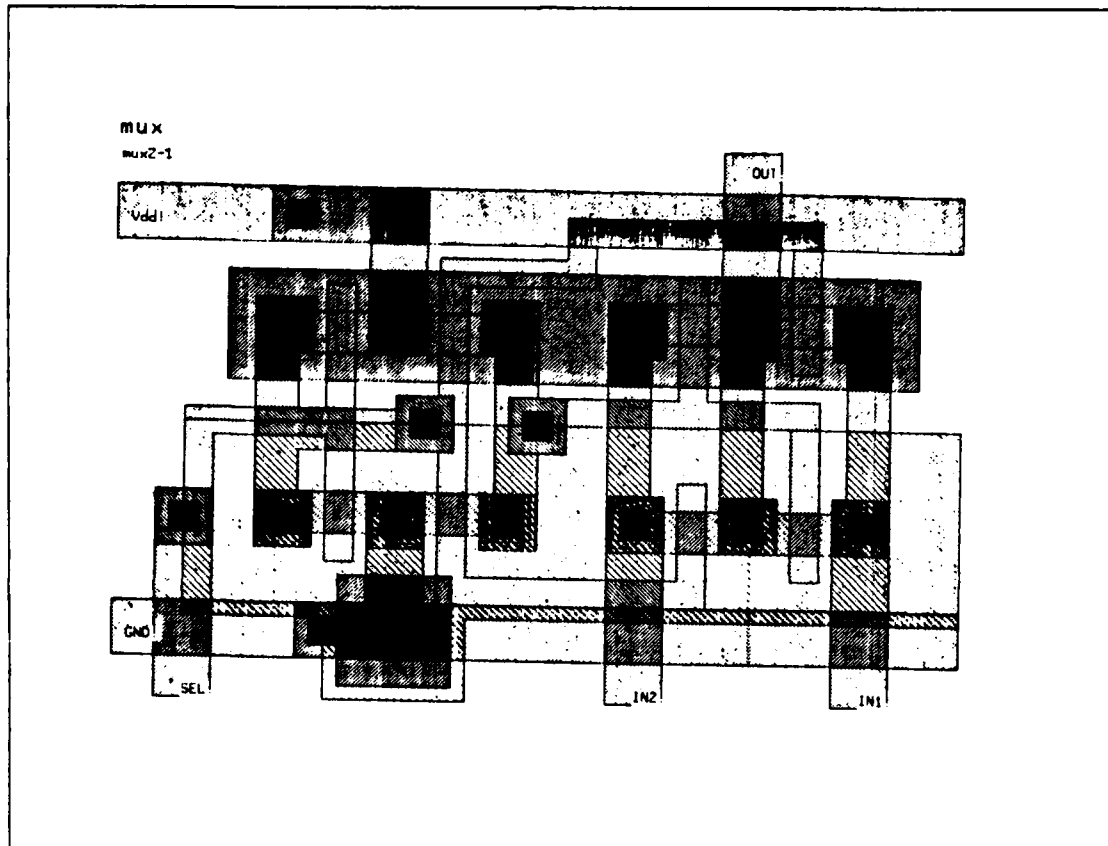


Figure 4.16 SCMOS 2:1 Multiplexer.

The inputs to a multiplexer are specified by the datapath object specification. The control lines to the multiplexer are specified in the control section of the object file. For example, the data-path specification of a 4-bit port-internal unit constructed of a multiplexer is:

```
(port-internal sequencer-light-controller-next-state -2
  (((constant 0)) ((constant 1)) ((constant 2)) ((constant 3))))
```

which would translate into the hard-wiring of four constant inputs to a 4:1 multiplexer. The multiplexer is integral to the data-path architecture and requires careful consideration in the SCMOS version of MacPitts.

F. FRAME.L

The source program *frame.l* contains the layout functions for the wing and skeleton. A portion of the code in the function (*layout-skeleton*) was modified to allow a two-phase clocking scheme.

1. Two-Phase Clocking

The clocked circuits considered in MacPitts are based on a three-phase non-overlapping clock signal. A non-overlapping two-phase clock is considered [Ref. 2] and implemented in the design of a SCMOS two-phase master-slave D flip-flop. To utilize this register, the third clock and its supporting wiring were removed.

To remove the specification of a *phic* clock in the compiler, the datapath and control extraction function (*extract-component-list*) was changed. This function specified the making of the three-phase clock. The lookup function in the program *general.l*, (*lookup-phic-pin#*) looks up the *phic pin#* in the input program. This function was removed along with the (*make-pin (lookup-phic-pin# definitions)(make-phic-pad)*) line of code in the (*extract-component-list*) function in *extract.l*. The clock wires run the channel between the data-path and control module. The (*layout-skeleton*) function generates three clock wires horizontally through the length of the internal-layout. To remove the *phic* clock wire, the layout geometry of that wire in the function (*layout-skeleton*) was removed. A slight, but relatively insignificant, reduction in chip area was achieved using a simple circuit design. The removal of a pad, however, promises significant savings with the insertion of the very large SCMOS pads. The reduction in the number of clock wires to be routed did not significantly affect computation time.

G. GENERAL.L

The layout information on the NMOS superbuffers and the river router are contained in the source program *genera.l*.

1. SuperBuffers

The four types of NMOS superbuffers (inverting/noninverting and inverting-pair/non-inverting-pair) are located in *general.l*. The SCMOS library contains a 1x and 4x drive non-inverting buffer. The 1x driven non-inverting buffer replaces the NMOS non-inverting buffer. The 4x driven non-inverting buffer replaces the NMOS non-

inverting-pair-super-buffer. SCMOS inverting buffers may use the SCMOS inverters. as a buffer.

2. River Router

The river router function (river) is defined in general.l The allowable NMOS layers for river routing are metal, diffusion and polysilicon. SCMOS layers added were metall1 and metal2, diffusion and polysilicon. The present river router routes polysilicon wires between the control and data-path units and routes diffusion wiring between the multiplexers and organelles in the data-path.

H. LAYER CONVERSION

Conversion of the NMOS layers in data-path.l and frame.l to SCMOS layers requires consideration of new routing schemes using the dual metal capability of SCMOS technology. A one for one exchange of SCMOS layers for NMOS layers resulted in the power and ground rail attachments to the skeleton being pushed into the power and ground pads, and so further modification is necessary.

I. SUMMARY

The results of the numerous modifications and insertions of SCMOS cells are illustrated in the taxi meter chip generated by the MacPitts program in Table 15. Figure 4.17 illustrates the change in the overall size and growth changes in the datapath from NMOS technology to SCMOS technology.

An expanded topological layout of the taxi data-path is given in Figure 4.18 The object file data-path specification given in Table 16, is mapped directly into the data-path along with its buses. The only NMOS organelles left in the SCMOS taxi chip are the incrementer and multiplexers. The ports are easily converted to SCMOS layers and SCMOS contacts. The ports are subject to revision when a better routing scheme is implemented.

It can be seen that the SCMOS version has grown considerably in width but is shorter in length. The SCMOS organelle designs are short in length and tall in height. The NMOS organelle designs were long in length and short in height.

Several one bit cells were also processed into MacPitts chips. The logical cells maintained approximately the same chip size and computation time as their NMOS counterparts. The significant changes in area occurred with the adder and register. The adder decreased the length of the chip significantly while the register increased the

TABLE 15
TAXI METER INPUT PROGRAM

```
(program taxi 8
  (def 17 power)
  (def 1 ground)
  (def 2 phia)
  (def 3 phib)
  (def 4 phic)
  (def timer register)
  (def fare register)
  (def reset signal input 5)
  (def time-on signal input 6)
  (def hire signal input 7)
  (def mile-mark signal input 8)
  (def display port tri-state (9 10 11 12 13 14 15 16))
  (def charge-time signal internal)
  (def maximum-time constant 100)
  (def base-fare constant 20)
  (def cost-per-mile constant 50)
  (def cost-per-time constant 10)
  (process time-clock 0
    off
      (cond (time-on (setq timer 0) (go on))
            (t (go off))))
    on
      (cond (time-on (cond ((= timer maximum-time)
                          (setq timer 0)
                          (signal charge-time))
                        (t (setq timer (1+ timer)))))
            (go on))
            (t (setq timer 0) (go off))))
  (process fare-clock 0
    for-hire
      (cond (hire (setq fare base-fare) (go hired))
            (t (go for-hire)))
    hired
      (par (cond ((not hire) (go for-hire))
                ((and charge-time mile-mark)
                 (setq fare (+ (+ fare cost-per-mile) cost-per-time))
                 (go hired))
                (charge-time
                 (setq fare (+ fare cost-per-time))
                 (go hired))
                (mile-mark
                 (setq fare (+ fare cost-per-mile))
                 (go hired))
                (t (go hired)))
          (setq display fare))))
```

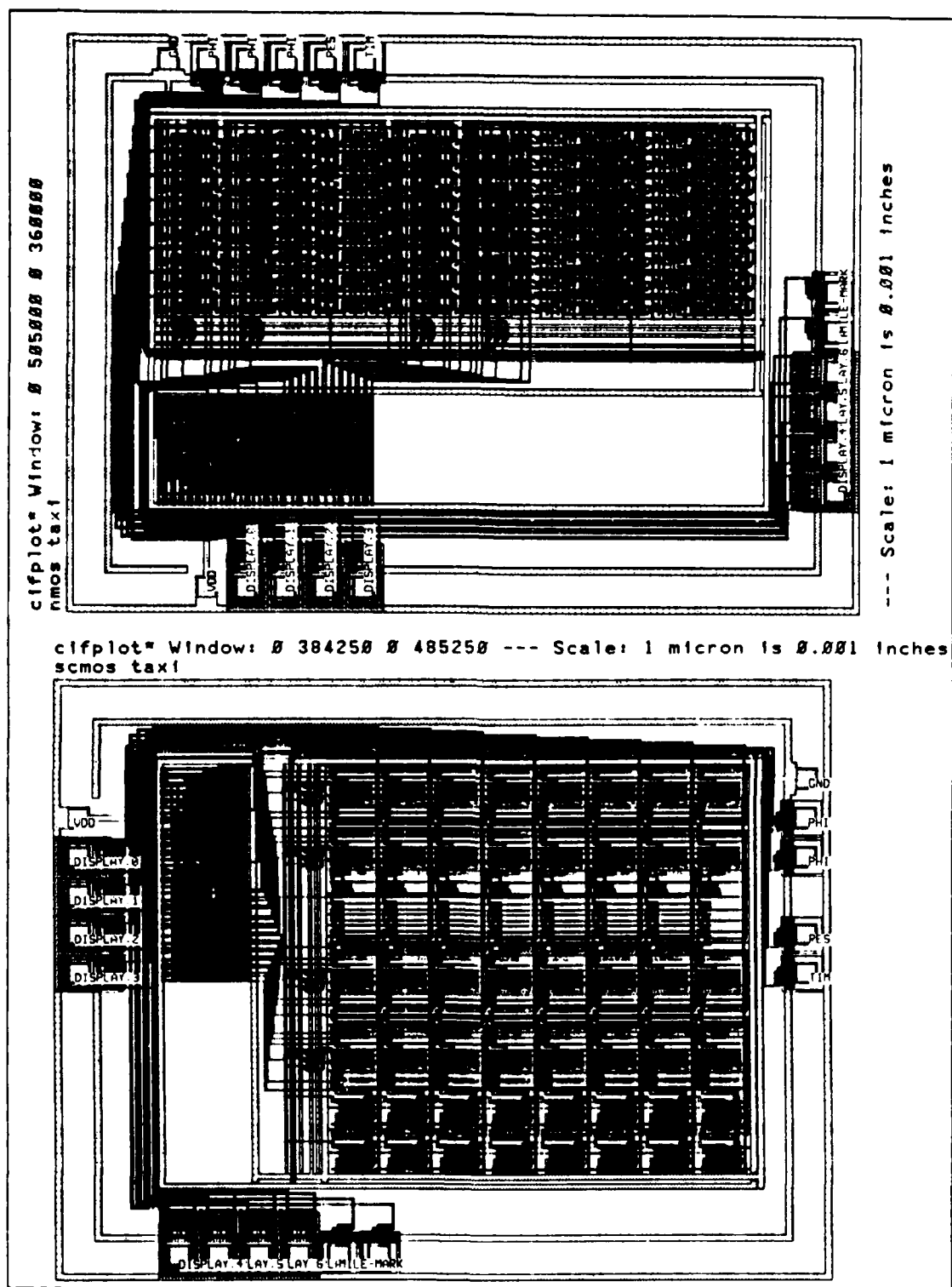


Figure 4.17 Comparison of an NMOS and SCMOS Taxi Meter Chip.

TABLE 16
TAXI CHIP DATAPATH SPECIFICATION

```
((register sequencer-time-clock-state -1 (((constant ))) ((internal 2))))
(port-internal sequencer-time-clock-next-state -2 (((constant 1)))
((constant 0))))
(bit (0) (((internal 1))))
(register timer -3 (((constant 0))) ((internal 4))))
(organelle = 0 (((internal 3) (constant 100))))
(organelle 1 + -4 (((internal 3))))
(register sequencer-fare-clock-state -5 (((constant 0))
((internal 6))))
(port-internal sequencer-fare-clock-next-state 6 (((constant 1))
((constant 0))))
(bit (0) (((internal 5))))
(register fare -7 (((constant 20))) ((internal 9)) ((internal 8))))
(organelle + -8 (((internal 7)) ((constant 50)) ((internal 7)
(constant 10))))
(organelle + 9 (((internal 8) (constant 10))))
(port-output display (((internal 7))))
```

width of the chip considerably. The taxi chip shows the reduction in length and increase in width. Table 17 summarizes the SCMOS cells currently available in the Mullarky SCMOS Cell Library along with the cells used by MacPitts and their corresponding library functions. The basic SCMOS combinational logic cells, an adder, and a static storage master-slave flip flop (register) have been installed in MacPitts. The cells remaining to be designed are flagged with a star *. These include an incrementer, decrementer, subtractor and comparator $<>$, $<>0$, $=$ and $=0$. The xnor cell requires a wired-and output to be useful as an $=$ test comparator.

The boolean functions also require SCMOS designs. Boolean type objects are implemented in the control and flags blocks. Unlike the integer types that are stored in registers or loaded onto ports, boolean types are stored in flags or propagated as signals. Boolean forms have a true or false result. The supported Boolean functions include logic cells, shift cells and comparison cells (*not*, *nor*, *and*, *or*, *equ*, *xor*, *parity*, $<<$, $>>$, $=$, $=0$, $<>0$, <0 , $<=0$, >0 , $>=0$, *unsigned- $<$* , *unsigned- $<=$* , *unsigned- $>$* , and *parity*).

The data-path has an internal set of organelles that are not called directly by the input program. The 'process' construct implies the use of a register to store state. In addition, the definitions statements in the input program contain I/O ports that involve construction of input, output and internal port structures. The bit form is simply a wire that is useful for extracting fields of data words. Bit returns a Boolean value of true or

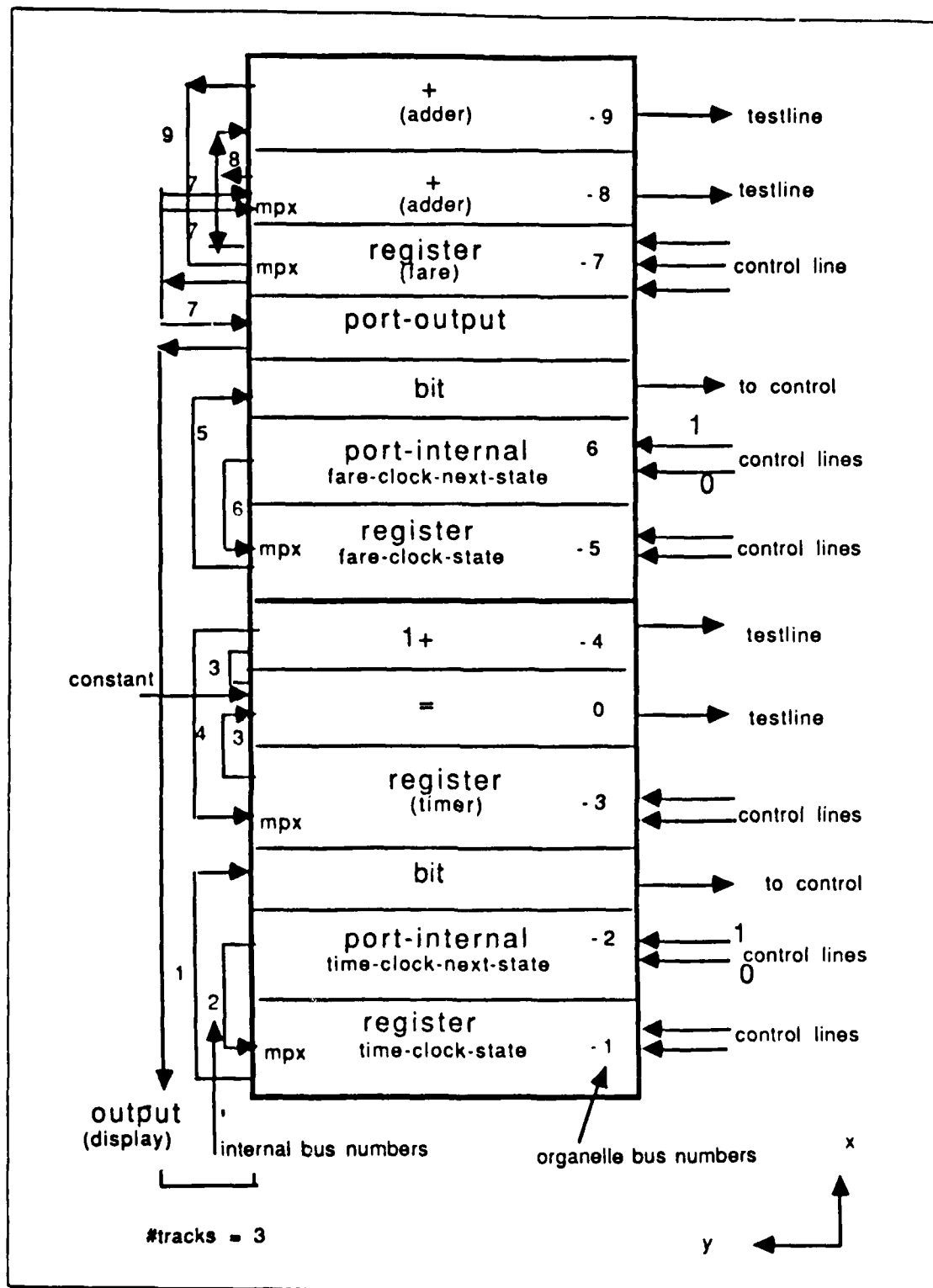


Figure 4.18 Topological Layout of the Taxi Chip Data-Path.

false. Data selectors or multiplexers are generated by control. The control section is under modification to a microprogrammed controller necessitating a multiplexer design to fit this controller's specification. [Ref. 3]

While the non-inverting buffer was replaced by the SCMOS buffer the inverting buffer has not yet been replaced. The enlarged two-phase register and the reduced size of the adder provide the largest changes in dimensions. The remainder of the organelles remain closer in sizing and computer processing time. Finally, the SCMOS contacts were designed by J. Harmon and are available on the ISI graphics editor.

TABLE 17
SCMOS CELLS AND MACPITTS REQUIREMENTS

SCMOS CELL LIBRARY	MACPITTS ORGANELLE LIBRARY	LIBRARY FUNCTION
xor2	layout-xor-organelle	word-xor
and2	layout-and-organelle	word-and
nand2	layout-nand-organelle	word-nand
nor2	layout-nor-organelle	word-nor
or2	layout-or-organelle	word-or
inv1x inv4x inv8x	layout-inverter-organelle	not
adder	layout-+organelle	+
*	layout-=organelle	=
xnor2	layout-equ-organelle	word-equ
*	layout-1+organelle	1+
*	layout-1-organelle	1-
*	layout-subtract-organelle	-
*	layout-<>organelle	<>
*	layout-<>0organelle	<>0
*	layout-=0organelle	=0
*	lsh-zero/lsh2/lsh3/lsh4/lsh8	<<
*	rsh-zero/rsh2/rsh3/rsh4/rsh8	>>
----- Boolean Organelles Functions -----		
	not nand nor and	
	or xor equ	
----- Data-Path Organelles -----		
dff1phase		
dff2phase	layout-register1	
*	layout-driver1	
*	driver-super-buffer	
mux2-1(not used)	layout-odd-operand	
*	layout-even-operand	
*	port-output1	
*	bit	
----- General Organelles -----		
buffer1x	layout-non-inverting-super-buffer	
buffer1x-4x		
*	layout-inverting-super-buffer	
----- L5 contacts -----		
*	m2c pc ndc pdc nsc psc	

NO-A184 873

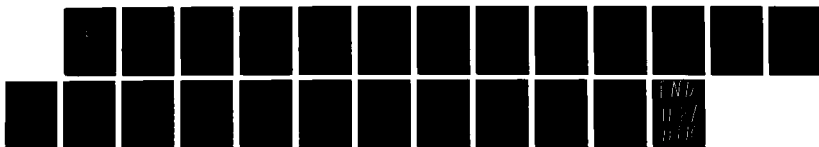
TECHNOLOGY UPGRADE OF A SILICON COMPILER(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA E G HALAGON JUN 87

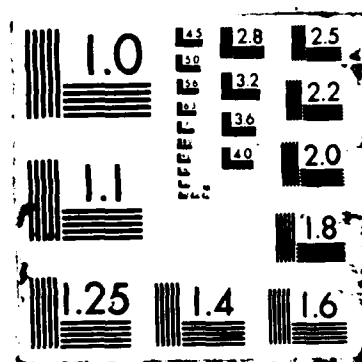
2/2

UNCLASSIFIED

F/G 12/5

NL





V. AREAS FOR FURTHER DEVELOPMENT

A. SUGGESTED CHANGES TO SOURCE PROGRAMS

Several additions and improvements are needed to modify the MacPitts silicon compiler. Chapter IV singles out the additional SCMOS cells to make the library complete. Mentioned here are areas where modification and new research can contribute to the implementation of an efficient MacPitts silicon compiler. Since changes to the compiler involve code changes, each source program is reviewed for changes accomplished and proposed changes to be implemented.

1. *lincoln.l*

The source program *lincoln.l* is akin to a writer's dictionary of user defined functions that extend the basic LISP primitives available in the Franz Lisp environment. This tool forms the basis of the MacPitts source programs. Before any other program will run, this file must first be loaded. The documentation of *lincoln.l* is contained in [Ref. 1.] This program contains the *cfasl* routine to convert the program *c-routines.c* into LISP code for use with the MacPitts interpreter. A problem arose with this section of the program with multiple definitions of the C functions in *c-routines.c*. This problem was resolved by removal of a twice defined variable in Franz Lisp called *ospeed*. Reference 3 discusses the installation of MacPitts in more detail. MacPitts was written in the early 1980's, and Franz Lisp has now incorporated many of the functions that are contained in *lincoln.l* in its symbol table. At some future date, implementing the more efficient Franz Lisp primitives that correspond to functions defined in *lincoln.l* in the MacPitts code will eliminate many calls to functions and speed up processing time.

2. *L5.l*

The source program, *L5.l*, is the heart of the layout language used throughout the MacPitts layout routines. It is akin to a technology file, in that it maintains global technology variables such as type technology (NMOS, SCMOS, CMOS-PW ...), sets feature sizes (200/250 centimicrons-per-lambda for NMOS processes), and defines the list of allowed-layers for each type of technology: ((NMOS: ND NM NP ...) (SCMOS: CAA CPG CMF CMS ...)).

The changes made at this level were discussed in Chapter III. A global switch to select NMOS, SCMOS or HYBRID technology was implemented. [Ref. 3] The NMOS contacts or vias are contained in this file. SCMOS contacts were added to this file. [Ref. 3.] The conversion routines are also maintained in L5, that is, the conversion from an *item* to a CIF output. In the LBS ⁷ version, L5 contains an interface routine to CAESAR.

Much like the CIF layout language which calls other symbols to build a circuit, L5 contains a symbol call that is implemented by a Lisp macro *defsymbol*. A prime feature of L5 is that it defines layouts in terms of *rectangular geometry*. This choice of rectangular geometry for MacPitts layout code is influenced by the availability of the CAESAR layout editor which also uses rectangular geometry. An interface routine is available in the LBS source file L5.l. A change to Magic as the layout editor of choice for SCMOS designs shifted the priority to *box geometry* descriptions. The insertion into MacPitts of Magic produced SCMOS designs necessitated a conversion from box (CIF) to rectangular (L5) geometry. For small scale SCMOS organelles, the conversion was not a problem. But for real life designs such as the SCMOS pads, the conversion proved unfeasible. The use of recursion in LISP to process CIF files of up to 940 lines of code caused a namestack overflow condition. Furthermore, the conversion of a *defsymbol* to an *item* creates a load on the system's computation time. The attributes that need to be passed to the *item* *defstruct* are the dimensions of the organelle (*left bottom right top*) and labels (*points*) identifying its terminals. A program to extract this information from the CIF file while leaving the file in CIF form saves in overhead and produces a compiler capable of handling large sub-structures such as the pads. This aids the ability to design more complex chips in shorter time, and makes it possible to place pads on four sides of the chip rather than three. A data structure to perform this extraction is described in Reference 3.

3. Defstructs.l

The source program *defstructs.l* contains the data structures used throughout the MacPitts source code. The *defstructs* listed in this program parallel the circuit symbols necessary to build a chip. The *defstruct object* provides the framework of the .obj file. The *definition* *defstruct* provides the translation of the input (.mac) program into the object (.obj) file specification. *Defstructs* provide the database that allow for

⁷The Lincoln Boolean Synthesizer (LBS) is a CMOS based silicon compiler using the same layout tools that MacPitts uses. It is documented in [Ref. 1] for combinatorial logic.

the creation, selection, mutation and interrogation of constructs that have a corresponding hardware implementation. To understand how MacPitts organizes the specification and layout of a module, or portion of a module, the root and children nodes of the defstructs pertaining to that module or sub-element need only to be traced.

The MacPitts NMOS controller uses a Weinberger array. The SCMOS version of MacPitts will use a microprogrammed controller that requires its own defstructs to specify its construction. A familiarity with how the Weinberger array is specified and an understanding of the coding of the Weinberger array in the source program *control.l* will facilitate the coding of the SCMOS microprogrammed controller.

The *data-path* related defstructs were discussed in Chapter 3. The purpose was to develop a methodology for tracing through the specification and layout language to determine the layout scheme of MacPitts. The same methodology can be applied to the other structural components that form a MacPitts circuit. For example, the layout of the pins, nets, buses, flags and sub-components such as the sequencers and multiplexers can be understood by tracing the coding that generates them. From this understanding, modifications can be easily applied. Thus, a thorough understanding of the source code in *lincoln.l*, *L5.l* and *defstructs.l* is required before modification to MacPitts should be attempted.

4. Library

The Backus Naur Format (BNF) grammar can be used to define the syntax of the MacPitts design language. The compiler converts the input syntax to an object file using the *definition* defstruct in *defstruct.l*. Four specific cases of the *definition* defstruct are loaded at run time in their expanded form. These constructs are *organelle*, *function*, *test* and *macro*. They are predefined in the *library* and are loaded at run times into the macpitts environment. For example, the design language functionally defines the arithmetic/logic grammar. That is, a BNF *<form>* can be a (*<function-name>* [*<form>*]*) like (*word-nand a b*) or a test form (*<test-name>* [*<form>*]*) as in (= in 5). The library contains the functional definition of *word-nand* in the (*function*) construct. The language form, *word-nand*, is passed to the *organelle* data structure, via the *function* construct in the *library* as *nand*. The *organelle* function in the library contains a call to the actual layout of the *nand organelle* contained in the source program *organelles.l*. The actual layout function is defined as (*layout-nand-organelle*). The *test* function is also an interface between the design language and the actual

implementation of the organelle. The syntax = is passed to the *test* function which then passes it to the organelle function =. The organelle function interfaces the syntax to the actual layout. The call to the layout in the *organelle data structure* is of the form (*layout* = *organelle*).

New organelles can be easily added to the library by placing the geometric description of the organelle in the source program *organelles.l* and functionally defining the *organelle* in a *function* or *test* form. An *organelle data structure* is also created by the program.

5. Organelles.l

The layout geometries of the organelles are contained in *organelles.l*. The SC MOS conversion of this library is incomplete. Cells need to be designed to replace the NMOS incrementer (1+), decrementer (1-), subtracter, =, < >, < > 0, and = 0 and shift organelles. Unlike the arithmetic/logic cells whose layout information is contained in the *gen-form* field of the organelle data structure in the file *library*, the *shift organelles* has both its layout and *gen-form* functions located in *organelles.l*. The arguments to the shift *gen-form* are (*info bit word-length direction distance fill drive ratio*). *Direction* can be *right* or *left*. *Distance* is either 1, 2, 3, 4 or 8 shifts as required. *Fill* is either *control* or *zero*. These arguments are passed by the *organelle data-structure* in the *library* to a function in *organelles.l* called (*query-handler-for-shift1-organelle*). A SC MOS shift design is required along with an understanding of the hierarchical construction and call of the NMOS shift organelle currently in the library.

6. Data-Path.l

The data-path consists of horizontal buses and an array of units. The units utilize built-in organelles contained in this program. The three-phase NMOS register defined in the function (*layout-register*) has been replaced by the SC MOS two-phase register. The remaining NMOS built-in organelles: *port-output*, *bit*, *driver*, *driver-superbuffer* require implementation in SC MOS. The driver in particular must be aligned to the register's clock lines. The bit wire is a simple change to the SC MOS layers CMF or CMS in the (*layout-bit*) function. The (*layout-port-output1*) function requires the change from the NMOS diff-cut contact to a SC MOS m2contact.

The NMOS multiplexer is hierarchically constructed from basic N-channel transistors in *data-path.l*. If the *mpx* specification in the object file contains several *operands* of the form (*constant number*), a multiplexer is generated at the inputs of the organelle or register. If one or no constant *operands* is specified, single metal wires are

generated at the inputs of the organelle. The multiplexer is specified by the requirements of the controller. Since an SC MOS microprogrammed controller is replacing the NMOS Weinberger array, the primitives required to construct a multiplexer to meet the needs of the controller specification are specified in Reference 3

The organelles are stretched to meet the power, ground and bus lines. These extensions, as well as the bus lines, require routing in the SC MOS dual metal layers. The buses are routed in first metal by changing the layers in the (*layout-buses*) function in MacPitts. The code that generates the layout of the buses and the segmentation and compaction algorithm require documentation.

7. Frame.1

The top-level layout function (*layout object*) is contained in this file. The coordination of the entire chip layout is performed by this single function. It generates the river routed polysilicon wires from control to the data-path and flags blocks. It lays the metal skeleton that distributes power and ground to the internal layout. It generates the I/O wiring called *wing* from the control to the edges of the internal layout for connection to the pads. This wiring is generated by the wing layout routines.

The basic floorplan generated by this function is a box within a box type of plan. The pins are placed on three sides with a ground and power ring connecting them to the internal layout. The internal-layout is positioned in the center of the chip. The area between the pins and the internal layout is called the ring. The wiring nets are located in this ring channel which connects the pads located on the top, right and bottom sides of the chip to the terminal points located only on the left edge of the internal-layout. The dimensions of the chip are fixed in the top and right sides and grows on the bottom and left sides. The problem with this scheme is that the largest module in the internal-layout determines the size of the invisible box around which a channel is formed for routing of wires from the pads. The insertion of the SC MOS two-phase registers extended the width of the circuit and shortened the length of the circuit. However, if the pads extend beyond the length of the internal layout, the channel is formed around an invisible box whose dimensions are determined by the longest element in the x and y direction. The pads or the internal layout determine the box dimensions. The number of pads along these edges can create a larger length than the length of the data-path causing unused silicon space. The problem of routing the nets becomes a problem of placement.

An important modification to the floorplan is the placement of the pads on four sides of the chip. A modification to the function (*pins-dimensions*) to place pads on four sides resulted in a reduction of chip size of a simple circuit by 30%. The algorithm specified in the (*pins-dimensions*) function determines the *maximum-#pins-horizontally* by dividing the length of the internal-layout A or *intended-right* location by the *pad-class-width*. For a 2.5 micron per lambda NMOS feature size, the pad width is 100 lambda and height is 82 lambda. The 2.0 micron per lambda NMOS feature size has a pad width of 128 lambda and a height of 112 lambda. As lambda shrinks pads become relatively larger. The SCMOS pads have a width of 198 lambda and a height of 425 lambda. This will have a significant impact on the chip size and placement routine. In light of the larger pads, the need for the placement of pads on all sides is evident.

The (*place-pin*) function determines the number of pins per side by finding the highest pin number and dividing it by the value 3. The function (*pins-dimensions*) determines the *top*, *right*, *bottom* and *left* dimensions of the chip. The pins are oriented around the three sides of the chip by the (*place-pins*) function. The functions (*extend-right*), (*extend-top*) (*pins-dimensions*) and (*side-extension*) contain the variable 3 which limits the placement of the pads to three sides of the chip. This value was altered to 4 to attempt to place the pins on four sides.

The (*place-pins*) function is capable of placing the pins on four sides of the chip. The algorithm that limits the pins to three sides is contained in the function (*pins-dimensions*). This function determines the number of pads per side as designated by *#pins-per-side* variable in the code. The pin specification in the object file is read and the highest numbered pad is set to the variable x. The variable y is set to the value of 3. The function (*/up*) takes the quotient x / y and applies the lisp function *fix* to the result. The Franz Lisp primitive *fix* returns a fixed number as close as possible to the number and rounds down. For example a chip with 13 pads and $y = 3$ would yield the value 4. This value multiplied by 3 yields 12 which is not equal to 13. Therefore, the value of 4 is incremented to the value 5. The *#pins-per-side* is assigned the value of 5.

The (*place-pins*) function contains an algorithm for pad placement. If the pad number referred to as the variable *pin#* is less than the variable *#pins-per-side*, i.e., less than or equal to the value 5, the pads are placed on top aligned to the left edge of the internal layout at $x = 0$. If the variable *pin#* is between the value 5 and two times the *#pins-per-side*, i.e., 10, the pads are placed on the right aligned to the bottom edge, at $y = 0$, of the internal-layout and stacked vertically. If the variable *pin#* is between the

value 10 and three times the *#pins-per-side*, i.e., 15, the pads are placed on the bottom aligned to the left edge of the internal layout at $x = 0$. If the variable *pin#* is greater than the value 15 and less than four times the *#pins-per-side*, the pads are placed on the left edge of the chip aligned to the bottom of the internal layout at $x = 0$. A division by the value 3 in the function (*pins-dimensions*) ensures that pads are never placed on the left edge of the circuit. A division by the value 4 will allow pad placement on all four sides. To illustrate, for a circuit with thirteen pads, the variable *#pins-per-side* is 4, allowing four pins on top, four pads on the side, four pads on the bottom and one pad on the left edge. The problem at this point is a problem of wiring the pads to the internal layout.

Placing pads on four sides will require changing the net list that routes the wires from the pads to the internal layout on the left side. The extraction of the signal netlist is done by the (*extract-nets*) routine contained in *frame.l*. Points at the left edge of the internal layout and at the pads are assigned any of the following attributes: *inside*, *outside*, *top*, *bottom*, *left*, *right*, *first*, and *last* which are used by the net functions to route the ring of wiring around the internal-layout.

8. General.l

This program contains the river routing algorithm that is used to route poly wires between the control and data-path/flags module. It is also used to route short diffusion layers between a multiplexer and an organelle. The SCMOS layers, diffusion (CAA), polysilicon (CPG) and metal1 and metal2 (CMF, CMS) were added to the river router. The choice of routing layers using the dual metal capability of SCMOS technology should be applied to the wiring generated by the router.

9. Extract.l

The data-path and control extraction as well as the sequencer, pins, flag, and definitions extraction are done in this program. The PHI-C pin was removed from the function (*extract-component-list*). The control extraction process requires further modification in order to replace the extracted specification to match the SCMOS microprogrammed controller.

10. Prepass.l

The highest level procedure (*macpitts-compiler*) located in the source program *prepass.l*, accepts the input program to be compiled and returns a list representing the hardware parameters to be implemented in the form of an object file. It also returns a geometric description of the layout in the form of a .cif file. The subsidiary procedure

(*get-object*) in *prepass.l*, accepts an argument list to be compiled and returns a list representing an object. The object is passed to the procedure (*layout object*) in *frame.l*, which then returns a list in L5 item format. The list or *item* is passed to the function (*cifout*) in the program *L5.l*, and converted to a CIF file. The compiler function sets the allowable *minimum-feature-size*. This option was expanded to accept a 3 micron feature size. Interface routines to the Magic layout editor were incorporated at this level [Ref. 3].

11. Control and Order.l

Order.l performs optimization functions in the ordering of the control unit's nor gates and the ordering of the units in the data-path. The optimization of the control and datapath internal order requires documentation. These programs are discussed in [Ref. 3].

12. Pads.l

The internal-layout forms the framework around which the pins and net wiring are formed. The (*layout-power-ring*) and (*layout-ground-ring*) functions in *frame.l* include vertical wire extension to the *intended-top* location of the internal-layout. The ground and power pads contain the side on which they are located as an attribute, i.e., top, right, bottom, left.

The NMOS pad library contains pad definition functions that call the layout geometry of the pads. The pad geometries are defined as defsymbols and the pad definitions call the defsymbols. Marks are generated for wire connections and for naming of the pads on the CIF outputs from the circuit. The input pad contains a mark for an in-wire with the attributes 'ring side 'outside. It contains an external attribute for the name of the cell to which it is to be connected. The output pad contains the same marks, an out-wire and an external name. The I/O pad has a mark for an in-wire, out-wire, drive-wide and an external for name. The clock pads are marked for clock-wires and external designation. Ground and power pads are marked for power and ground connection points and for external 'vdd and 'gnd. These same pad-definitions must be inserted into the SCMOS pad definitions.

NMOS pads are contained in two files, *rinout* for a minimum feature size of 5 microns and *pad20b*, supporting a minimum feature size of 4 microns. These files are large CIF files that are converted to defsymbol form by the program *padgen.l*. The results of this conversion routine are loaded into the file *pads.l*.

The change to SCMOS requires the insertion of SCMOS pads at 3 microns minimum-feature size. The 2 micron NMOS pads measure 100 lambda on a side. As lambda shrinks pads must become relatively larger, unless the technology for bonding wires to pads improves as well. The pads are large to allow a thin wire to make contact to them. In the case of the SCMOS pads available in the *vlsi* directory, the pads are significantly larger than the NMOS pads. The SCMOS I/O pads take up to 940 lines of CIF code. Converting this size program to *defsymbol* form caused a *namestack* overflow condition. The routine that performs the conversion is recursive and cannot hold enough information on a stack. The SCMOS Vdd and Gnd pads, which require less storage, were converted to *defsymbol* form and inserted into the *macpitts* environment. Incorrect scaling information caused the pads to overlap into the internal layout. The correct height and width of the pad at 1.5 micron per lambda should correct the overlapping problem.

The ground pad must be on the top side of the circuit and the power pad cannot be on the top side of the circuit. The power pad is generally located on the bottom side of the circuit. These definitions are contained in the (*layout-ground-ring*) and (*layout-power-ring*) functions. The pads placed on the top side are the first third of the total pads on the circuit and are identified by their pin numbers. Ground is usually the first pin. This pad is mirrored around the x-axis. It is placed on the circuit by the (*place-pin*) function and instantiated by the (*layout-pad*) function in *frame.l*. The power pad normally located on the bottom of the circuit is not translated in orientation. The SCMOS power and ground pads were inserted into *MacPitts*. Their large size overlapped will into the circuit due to inaccurate spacing information. The associated wiring was not instantiated because the test case failed to insert marks to identify hook-up points to the wiring.

The functions that were modified were the functions: (*pad-class*) , to allow a 3 micron minimum-feature-size, a (*pad-class-default-power-bus-width*) of 40 lambda for SCMOS, a (*pad-class-basic-height*) of 425 lambda, a (*pad-class-width*) of 198 lambda, and a (*pad-basic-extension*) of *nil*. As a quick test, the *rinout power pad* function was used to place the SCMOS pad into the *MacPitts* generated circuit. This was done with a pad cell definition of the form

```
(defun layout-rinout-ground-pad (power side)
  (merge (PadVdd) (mark 'power 50 76 'CMF (list side)
    (mark 'vdd 50 53 'CMF (list 'external side))))))
```

and the insertion of the `defsymb` containing the geometry of the power pad of the form

```
(defsymb PadVdd
```

```
  nil
```

```
  (merge (rect 'CWP 10 23 16 27) .....))
```

were inserted into the `macpitts` environment. A `MacPitts` program was executed to test the effect of these changes. The correct sizing information and correct position for connection of the wiring extending from the pad requires continued work to allow the insertion of SC MOS pads into `MacPitts`.

The power and ground pads are not marked for connection to the ring network. Their wires connect directly to the internal layout frame. The frame is the *skeleton* composed of the ground, power and clock distribution for the internal layout. The *(layout-ground-ring)* and *(layout-power-ring)* functions in `frame.l` generate the ground and power buses through the pads and include the vertical wire extensions from the ground pad and power pad to the skeleton. The ground and power pad contain the side on which they are located as an attribute. For example, the ground wire extension originates from the location of the ground mark, coded as follows: *(mark 'ground x y layer (list side))* to the *intended-top* location of the internal-layout. The power extension wire originates from the marked 'power' location on the edge of the power pad *(mark 'power x y layer (list side))* to the lower edge of the internal-layout. This edge is the $y = 0$ line. The orientation of the cells is done by the *(place-pin)* function. If the pad is placed on *top*, it is *mirrorx*. If the pad is placed on the right is it *rotccw*. If the pad is placed on the bottom is is unchanged. If the pad is placed on the left edge it is *mirrorx* and then *rotccw*.

Insertion of the SC MOS input, output and I/O pads is not easily done using the `defsymb` scheme. The number of lines of CIF code is too large for LISP to handle. The `defsymb` form causes the creation of an item out of the geometry of a cell. The item generated contains the bounding box and points of the cell. The geometry of the cell is then converted back into CIF and accessed by a symbol-number. A routine is under development to extract the bounding box and points information directly from the CIF code without having to do the conversion. The new function is called *CIFS*YMBOL and is under development by J. Harmon. Until the *cifsymb* form is available the input, output and I/O pads cannot be inserted into `MacPitts`. The *cifsymb* is a significant change to the `MacPitts` compiler in that it

reduces computation time by a large factor. This improvement facilitates large chip designs in a reasonable amount of computer time.

13. **Flags.l**

Finally, the last remaining cells to be designed for SCMOS conversion are the cells contained in the `flags.l` source program. A number of defsymbols are contained in this program. They include *bottom-out-clock* and *bottom-out* which support the (*layout-flags*) function. The SCMOS design of the flags-clock, flags-power, flags-top-row, 2nd-row and 3rd-row is also required.

B. TECHNOLOGY INDEPENDENCE

The MacPitts prototype NMOS silicon compiler was heavily technology dependent. This made porting new technology into MacPitts tedious and difficult. The compiler should be written for technology independence. The fabrication technology description should be read from a technology file at the start of execution of the compiler. Programs would then access technology data exclusively through a data-structure built from this file. The MacPitts software programs could then avoid "hard-coding" technology layers allowing easy portability to new technologies. The L5 language acts as a technology file in the current version of MacPitts. The generic layers should be added to this file, thus removing layers from the layout routines.

C. ROUTING AND PLACEMENT

The area inefficiencies of the "box-within-a-box" algorithm is magnified when it is used with the SCMOS pads. The longest and widest element in the top level layout determine the channel wiring rectangles around the internal layout. An optimized internal-layout is overshadowed by the enlarged pads which add length and width to the circuit leaving enormous gaps of empty silicon area between the internal layout and the wiring or ring layout. A smarter placement algorithm requires investigation to optimize the channel wiring scheme. The first step in placement improvement is placing the pins on four sides. This will reduce the overall profile of the pads relative to the internal layout. Connections between the pads and the internal layout are at present only on the left edge of the internal layout. This causes the wiring on the left edge to bunch up and creates long bus wiring inside to the internal layout. The design of the organelles and registers is organized to fit into the array like placement of the units in that their inputs are taken from the left and their outputs from the top of the cell. New cell schemes are required and new optimization routines must be considered to place

the units in such a manner that they can connect on the right edge of the internal layout to the pads. This entails the overhaul of the bus layout routines, the net extraction and layout routines and the optimization routines in the source program *order.l*.

The MacPitts datapath can be partitioned into several independent processes and placed along with the flags block in an optimum arrangement relative to the control unit to reduce the size of the internal layout. A placement algorithm that sizes each block and places them in such a way as to optimize silicon area requires the use of artificial intelligence techniques such as best-fit or A* search routines. [Ref. 6] The datapath processes can be separated into mini-datapaths and placed relative to the control unit. This would save in silicon area over the current horizontal abutment of processes and the flags block. The fixed topology of the MacPitts generated datapath is basic and leaves much room for the implementation of smarter placement and routing schemes.

VI. CONCLUSIONS

The goal of this thesis was to insert SCMOS arithmetic/logic and memory cells into the MacPitts silicon compiler. Thus, the investigation serves as a case study on the conversion of an NMOS system to a custom SCMOS system, and provides a methodology for porting a new technology into the existing MacPitts compiler.

The addition of a SCMOS library is a small step toward the goal of a working SCMOS silicon compiler. Organelles and registers are not isolated elements but are part of a complete system that also must be designed with the capabilities of SCMOS technology in mind. The conversion of designs to SCMOS technology is not a straightforward procedure. Substantial alterations to the entire system architecture of MacPitts is necessary to obtain an efficient implementation. Chapter 5 details many of the changes required to the source code to accomplish an efficient SCMOS silicon compiler.

MacPitts as a software system hitherto had scarce documentation to facilitate alteration/modification to the source code to take advantage of rapidly changing technologies and smarter algorithms. The reduction of theory to practice in silicon compilation is paced by the availability of the necessary algorithms. It is taking time for the full capabilities offered by silicon compilation theory to be converted into practical and economical VLSI designs. The chief factor that makes silicon compilation tools possible is the introduction of artificial intelligence search routines for intelligent placement and routing. In Reference 7, two-layer channel routing algorithms are presented in which wires may run on top of each other for short distances as long as they are different layers. This would take advantage of SCMOS dual metal capability. A restricted two-layer wiring is presented in Reference 8.

Alternative methods of experimenting with MacPitts can be accomplished by taking the technology independent object file and entering portions of the specification into alternate programs for processing. Reference 5 documents the results of entering MacPitts technology independent specifications into another layout program. FAMOS is a standard cell placement and routing program in use at GTE Laboratories for layout of MOS integrated circuits. It places arbitrary sized cells in rows according to the strength of their interconnections and wires them using Hightower's algorithm. The

data-paths of the MacPitts chips were entered as cells in the FAMOS library, so that the existing I/O points would be wired automatically by FAMOS. A computer program was written to convert the control-logic portion of the object file to FAMOS format. The results of the two chip designs is documented in Reference 5

The general contribution of this thesis toward the goal of a SC MOS silicon compiler is in exposing the MacPitts code and hopefully shortening the learning curve necessary to enter MacPitts and make modifications. The basic algorithms for pad placement and circuit layout were uncovered from the code. The 'box-within-a-box' approach in laying out the top-level design coupled with the placement of pads on three sides of the circuit is wasteful of silicon area especially in light of the significantly larger SC MOS pads. The next step in the conversion process is placing the pads on all sides of the chip by exposing the wiring algorithms contained in the (extract-nets) function in frame.l. Making changes to a compiler without any documentation of its inherent algorithms slows the process of conversion to take advantage of new technologies, smarter algorithms for route and placement and new tools for program optimization to reduce computation and memory requirements.

The next step in this process is completing the SC MOS cell library and inserting it into MacPitts, inserting the microprogrammed controller, placing the pads on four sides by exposing the net layout algorithm and modifying it, inserting the SC MOS pads, scaling the SC MOS power bus by entering conductivity information of each SC MOS cell in the library, implementing a bus scheme that optimizes the use of dual metal wiring, and completing the wiring around the organelles' stretch connections to SC MOS layers. Following these tasks, the consideration and implementation of alternate placement algorithms and associated routing schemes can considerably reduce waste of chip area and reduce chip size.

From this point on, the consideration of artificial intelligence techniques for smarter algorithms and interactive generation of complex chip designs is limited only by the amount of time it takes the user to absorb the documentation of MacPitts as it now exists.

APPENDIX

CIFDEF.L

This file is currently not checking the layer-table for allowable layers. This feature will be added. When using this program the minimum-feature-size must be specified, by the command -> (setq minimum-feature-size '300) Once this program is inserted into L5.1 the minimum-feature-size is set globally. This file will not load into lisp on its own, it requires lincoln.l be loaded initially.

```
include front-page.l
```

```
;;Data-type
```

```
(defstruct read-cif-symbol
  (name program))
```

```
;;Command to convert a .cif file to a .L5 file
```

```
(defun cifsave (file)
```

```
  ;Converts an item from
```

```
  ;Cal Tech Intermediate Form (CIF) format to L5 code.
```

```
  ;E.g., (cifsave ' <file> ) where file is the file.cif without the .cif.
```

```
  ;The filename is quoted due to the lambda form.
```

```
  ;An nlambda form would not
```

```
  ;require quoting.
```

```
  (setq defsymbols (cif-in file))
```

```
  (setq output (outfile (concat file '.L5)))
```

```
  (pp-form defsymbols output)
```

```
  (patom "OK ")
```

```
  (close output))
```

```
(declare (special piport))
```

```
(defun cif-in (filename)
```

```
  ;E.g., (cif-in ' <file> ) returns a list in the form
```

```
  ;(defsymbol <filename> nil (merge (rect 'CPG 1 2 3 4)(.....)))
```

```
  ;The filename is quoted.
```

```
  ;Each cif symbol in manhattan geometry is converted
```

```
  ;to rectangular geometry (L5 form).
```

```
(let (cif-file defsymbols)
```

```
  (cond ((null? filename) (setq cif-file piport))
```

```
        (t (setq cif-file (infile (concat filename '.cif)))))
```

```
t))?[H      (t (setq cif-file (infile (concat filename '.cif)))))
```

```
  (setq defsymbols (make-defsymbol1
```

```
                    (cif-to-L5 cif-file () ())))
```

```
  (close cif-file)
```

```

    defsymbols))
(defun make-defsymbol1 (defsymbol)
  ;The argument 'defsymbol' is a data
  ;list in the format described by the
  ;defstruct 'read-cif-symbol'.
  ;From defsymbol, each of the defstructs
  ;arguments can be retrieved.
  ;Recall, a short defstruct selector is of the
  ;form- (type-field(i) 'field).
  ;This defstruct has two fields, name and
  ;program. To retrieve that information,
  ;the following selectors apply-
  ;(read-cif-symbol-name defsymbol) returns the name of the file.
  ;(read-cif-symbol-program defsymbol)
  ;returns the rectangular coordinates
  ;of the converted cif file.
  ;The conversion is done by 'cif-to-L5' function.
  °(defsymbol ,(read-cif-symbol-name defsymbol) ()
    ,(cons 'merge (read-cif-symbol-program defsymbol))))
(declare (unspecial piport))
(defun cif-to-L5 (cif-file defsymbols level)
  ;Parses the cif file and converts
  ;each line of coordinates
  ;which are in the bd ;(defun cif-to-L5 (cif-file defsymbols level)
  ; Parses the cif file
  ;and converts each line of coordinates which are in the
  ;manhattan geometry form
  ;(B length width xcenter ycenter) to rectangular
  ;geometry of the form (rect 'layer x1 y1 x2 y2). The results of this
  ;conversion is put into a list created by the
  ;read-cif-symbol defstruct.
  ;*****NOTE***** The Magic produced .cif files
  ;often are missing layers in
  ;the label (94) section of the .cif file.
  ;The default layer is polysilicon
  ;CPG and is inserted wherever
  ;the .cif file has no layer in its '94'lists.
  (setq command (read cif-file))
  (cond ((null command)
    (setq program (reverse defsymbols))
    (make-read-cif-symbol name program))
    ((eq command 'DS)
    (setq symbol-number (read cif-file))
    (patom (concat "Reading in symbol " symbol-number))(terpr)
    (setq scale-up (read cif-file))

```

```

                (setq scale-down (read cif-file))
                (cif-to-L5 cif-file defsymbols level))
((eq command '9)
  (setq name (read cif-file))
  (cif-to-L5 cif-file defsymbols level))
((eq command 'L)
  (cif-to-L5 cif-file defsymbols level))
  (setq new-level (read cif-file))
  (cif-to-L5 cif-file defsymbols new-level))
((eq command 'B)
  (setq length (read cif-file))
  (setq width (read cif-file))
  (setq xcenter (read cif-file))
  (setq ycenter (read cif-file))
  (setq x1 (centimicrons-to-lambdas (- xcenter (/ length 2))))
  (setq y1 (centimicrons-to-lambdas (- ycenter (/ width 2))))
  (setq x2 (centimicrons-to-lambdas (+ xcenter (/ length 2))))
  (setq y2 (centimicrons-to-lambdas (+ ycenter (/ width 2))))
  (cif-to-L5 cif-file
    (cons (list 'rect
      (list 'quote level) x1 y1 x2 y2) defsymbols)
    level))
((eq command '94)
  (setq label (read cif-file))
  (setq x (/ (read cif-file) minimum-feature-size))
  (setq y (/ (read cif-file) minimum-feature-size))
  (setq layer (ratom cif-file))
  (cond ((eq layer ' )
    (setq insert-layer 'CPG)
    (cif-to-L5 cif-file )
    (cons (list 'mark (list 'quote label) x y
      (list 'quote insert-layer) () ) defsymbols)
    level ))
    (t (cif-to-L5 cif-file
      (cons (list 'mark (list 'quote label) x y
        (list 'quote layer) () ) defsymbols)
        level ))))
((eq command 'DF)
  (cif-to-L5 cif-file defsymbols level))
((eq command 'C)
  (read cif-file)
  (cif-to-L5 cif-file defsymbols level))
((eq command 'End)
  (cif-to-L5 cif-file defsymbols level))
(t (patom "Warning - Invalid CIF Command ")
  (terpr) )))

```

(defun centimicrons-to-lambdas (value) (/ value minimum-feature-size))?-H
 (defun centimicrons-to-lambdas (value)
 (/ value minimum-feature-size))

LIST OF REFERENCES

1. Malagon-Fajar, M. A., *Silicon Compilation Using a Lisp-Based Layout Language*, M. S. Thesis, Naval Postgraduate School, Monterey, California, March 1987.
2. Mullarky, A., *SCMOS Cell Library for a Silicon Compiler*, M. S. Thesis, Naval Postgraduate School, Monterey, California, March 1987.
3. Harmon, J.E., *Automated Design of a Microprogrammed Controller for a Finite State Machine*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.
4. Froede, A. O., *Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.
5. Fox, J. R., *The MacPitts Silicon Compiler: A View from the Telecommunications Industry*, VLSI Design, May/June 1983.
6. Winston, P. E., and Horn, B. K., *LISP*, 2nd ed., Addison-Wesley, 1984.
7. Rivest, R. L., Baratz, A. E., and Miller, G., "Provably Good Channel Routing Algorithms", in Kung, Sproull, and Steele [1981], pp. 153-159.
8. Brown, D.J. and R. L. Rivest, "New lower bounds for channel width," in Kung, Sproull, and Steele [1981], pp. 178-185.

BIBLIOGRAPHY

- Kung, H. T., Sproull, R., and Steele, G., editors, *VLSI Systems and Computations*, Computer Science Press, 1981.
- Evanczuk, S., editor, *Results of a Silicon Compiler Design Challenger*, VLSI Design, July 1985.
- Southard, J. R., *MacPitts: An Approach to Silicon Compilation*, Computer, December 1983.
- Srini, V. P., *Test Generation for MacPitts Designs*, Proceedings IEEE International Conference on Computer Design, (ICCD-83), 1983.
- Crouch, K. W., *L5 User's Guide* Massachusetts Institute of Technology Lincoln Laboratory Project Report RVLSI-5, March 1984.
- Suzim, A. A., *Data Processing Section for Microprocessor-Like Integrated Circuits* IEEE Journal of Solid-State Circuits, Vol. SC-16, No. 3, June 1981.
- Barber, G. R., "Lisp vs. C for Implementing Expert Systems", AI Expert, Vol. 2, No. 2, February 1987.
- Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.
- Carlson, D. J., *Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.
- Foderado, J. K., Sklower, K. L. and Layer, K., *Lisp Manual*, University of California at Berkley, June 1983.
- Larrabee, R. C., *VLSI Design with the MacPitts Silicon Compiler*, M. S. Thesis Postgraduate School, Monterey, California, June 1985.
- MacLennan, B. J., *Principles of Programming Languages: Design Evaluation Implementation*, Holt, Rinehard and Winston, 1983.
- Lu, S., and Ousterhout, J. K., *Magic Technology Manual 2: Scalable CMOS Computer Science Division, Electrical and Computer Sciences, University of California*, 1985.
- Hill, D.D., Keutzer, K., and Wolf W., *Overview of the IDA System: A Toolset for VLSI Layout Synthesis*, AT&T Bell Labs, Murray Hill, New Jersey, 1986.
- Domic, A., *Silicon Compilers for VLSI*, Massachusetts Institute of Technology Lincoln Laboratory, May 1985.
- Pollack, S., Erickson, B and Maxor S., *Silicon Compilers Ease Complex VLSI Design*, Computer Design, September 1986.
- Bryant, R., editor, *Third Caltech Conference on Very Large Scale Integration*, Computer Science Press, 1983.
- Ullman, J. D., *Computational Aspects of VLSI*, Computer Science Press, 1984.

Southard, J. R., *An Introduction to MacPitts*, Massachusetts Institute of Technology Lincoln Laboratories Project Report RVLSI-3, February 1983.

Siskind, J. M., Southard, J. R. and Crouch, K. W., *Generating Custom High Performance VLSI Designs from Succint Algorithmic Descriptions*, MIT Conference on Advanced Research in VLSI, 1982.

Wilensky, R., *Lispcraft*, W. W. Norton & Co., 1981.

Mead C. and Conway, L., *Introduction to VLSI Systems*, 2nd ed., Addison-Wesley, 1980.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1
5. Dr. D. E. Kirk, Code 62KI Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	8
6. Dr. H. H. Loomis, Code 62LM Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	3
7. Prof. M. Zyda, Code 52MZ Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1
8. Mr. P. Blankenship Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, MA 02173-0073	1
9. Mr. J. O'Leary Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, MA 02173-0073	1

- | | | |
|-----|--|---|
| 10. | Dr. T. Bestul
Naval Research Laboratories
Code 7590
Washington, D. C. 20375 | 1 |
| 11. | Dr. D.O'Brien
Lawrence Livermore National Laboratory
P.O. Box 5504, L-156
Livermore, CA 94550 | 1 |
| 12. | Mr. A. DeGroot
Lawrence Libermore National Laboratory
P.O. Box 808,
Livermore, CA 94550 | 1 |
| 13. | Mr. H. N. Spelter
2918 Consaul Road
Schenectady, NY 12304 | 1 |
| 14. | LtCdr M. A. Malagon-Fajar
1200 Foursome Lane
Virginia Beach, VA 23462 | 1 |
| 15. | Mr. S. G. Spelter
1014 Ostrander Street
Schenectady, NY 12307 | 1 |
| 16. | Lieutenant J. Baumstark, USN
392-B Ricketts Road
Monterey, CA 93940 | 1 |
| 17. | Captain Gordon R. Steele, USMC
1166 Spruance Road
Monterey, CA 93940 | 1 |
| 18. | Captain E. G. Malagon
1200 Foursome Lane
Virginia Beach, VA 23462 | 1 |

END

11-87

DTIC